Data Compressions on Machines with Limited Memory

Debra Ann Lelewer

Department of Information and Computer Science
University of California, Irvine
Technical Report No. 91-36

Dissertation
submitted in partial satisfaction of the requirements for the degree of
Doctor of Philosophy in Information and Computer Science

Dissertation Committee

Professor Daniel S. Hirschberg, Chair

Professor George S. Lueker

Professor Michael B. Dillencourt

Dedication

This dissertation is dedicated to the memory of my father, Richard Fremont Evans, whose love and pride have inspired me and given me confidence all of my life.

Contents

List of Figures

List of Figures

List of Tables

List of Tables

Acknowledgements

I would like to thank the members of my committee, Daniel Hirschberg, Mike Dillencourt, and George Lueker, for their help, guidance, and encouragement during my graduate career. I am especially grateful to my advisor, Dan Hirschberg. His influence is reflected in my research and in my professional life.

The research on decoding prefix codes was joint work with Dan Hirschberg and was described in an article in the *Communications of the ACM* in April 1990. The work on context modeling will be presented at the *Data Compression Conference*, Snowbird, Utah, April, 1991.

My graduate studies have been supported by fellowships from the University of California and the California State University. I have received numerous grants from the Affirmative Action Faculty Development Program at California State Polytechnic University, Pomona. The College of Science at Pomona has also provided grants of travel funds and reassigned time.

I am grateful for the support and encouragement of many colleagues at the University of California, Irvine, and at California State Polytechnic University, Pomona. I thank Dennis Volper for serving on my candidacy committee and for assistance and support. I have had many enjoyable and rewarding discussions with Cheng Ng and Lynn Stauffer. I appreciate our research conversations and value our friendship. The support of many colleagues at Cal Poly Pomona has sustained me throughout my graduate studies. I am especially grateful to Norton Riley and Bruce Hillam for their friendship and concern. The interest and enthusiasm of hundreds of my students has served as a constant reminder of the importance of the goals I have pursued.

My most heartfelt thanks goes to my family. My husband, Steve, has been patient and encouraging; with the conclusion of the odyssey he is ecstatic. His pride is my most precious reward. My parents, Richard and Betty Evans, provided me all of the advantages and opportunities that they never had. My dreams have been their dreams and my achievements are their achievements.

Finally, I appreciate many good friends who have suffered my years as a social recluse with understanding.

Abstract of the Dissertation

Data Compression on Machines with Limited Memory

by

Debra Ann Lelewer

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1991

Daniel S. Hirschberg, Chair

We consider two problems in which machines with limited internal memory are used to compress and decompress data. In the first application, a powerful encoder transmits a coded file to a decoder that has severely constrained memory. A data structure that achieves minimum storage is presented, and alternative methods that sacrifice a small amount of storage to attain faster decoding are described. The second problem we address is that of encoding and decoding in limited memory. Methods for representing context models succinctly are described. These methods provide compression performance that is superior to state-of-the-art techniques, and competitive with newer approaches that use five times as much internal memory.

Introduction

Technology has made attractive the storage of enormous quantities of data on computer media and mass transfer of data over computer communication lines. The amount of data stored and transmitted daily is growing exponentially as computer use extends to more and more new disciplines and as computer communication networks proliferate. Data compression is the business of reducing the representation of information. Compressing data allows us to store information more compactly and transmit it more swiftly. Viewed in another way, compressing a data file to half its original size is equivalent to doubling the capacity of the media on which it is stored. It may then become feasible to store the data at a higher, thus faster, level of the storage hierarchy and gain the additional benefit of reducing the load on the input/output channels of the computer system. Compressing data to half its original size is equivalent to doubling the throughput of the communication channel along which the data is transmitted. The problem of compressing data as effectively as possible is a challenging one and the rich body of research on data compression algorithms provides evidence of both its importance and its complexity.

It is essential to recognize that the task of effective data compression is impacted by the resources available to be applied to the job and by the nature of the data to be compressed. The compressor and/or decompressor may need to execute in a limited-memory environment, or at a certain minimum speed. Approaches that work well when memory is abundant may not be effective in the limited-memory environment while methods that painstakingly analyze the data in order to determine how best to compress it are inappropriate for real-time applications. Our research deals with applications that limit the amount of internal memory and require reasonable execution speed.

Chapter 1 provides a framework for the study of data compression algorithms, including terminology, classification of methods, and measures of effectiveness. In Chapter 2 we describe existing data compression techniques to which our algorithms must be compared. We present research results in both of the component areas of data compression, modeling and coding. In Chapter 3 we present our solution to a specific data compression problem in which the model is fixed. Both use of memory and decode speed are constrained by the application. We design an implementation of the coding component that meets the

demands of the specific problem. Chapters 4 and 5 present work on general-purpose data compression algorithms that provide very good compression while having modest memory requirements and executing quickly. We employ a standard coding technique. Our contribution involves adapting a model to the demands of the application domain. Chapter 6 includes a summary of our contributions and suggestions for future research.

Introduction

Technology has made attractive the storage of enormous quantities of data on computer media and mass transfer of data over computer communication lines. The amount of data stored and transmitted daily is growing exponentially as computer use extends to more and more new disciplines and as computer communication networks proliferate. Data compression is the business of reducing the representation of information. Compressing data allows us to store information more compactly and transmit it more swiftly. Viewed in another way, compressing a data file to half its original size is equivalent to doubling the capacity of the media on which it is stored. It may then become feasible to store the data at a higher, thus faster, level of the storage hierarchy and gain the additional benefit of reducing the load on the input/output channels of the computer system. Compressing data to half its original size is equivalent to doubling the throughput of the communication channel along which the data is transmitted. The problem of compressing data as effectively as possible is a challenging one and the rich body of research on data compression algorithms provides evidence of both its importance and its complexity.

It is essential to recognize that the task of effective data compression is impacted by the resources available to be applied to the job and by the nature of the data to be compressed. The compressor and/or decompressor may need to execute in a limited-memory environment, or at a certain minimum speed. Approaches that work well when memory is abundant may not be effective in the limited-memory environment while methods that painstakingly analyze the data in order to determine how best to compress it are inappropriate for real-time applications. Our research deals with applications that limit the amount of internal memory and require reasonable execution speed.

Chapter 1 provides a framework for the study of data compression algorithms, including terminology, classification of methods, and measures of effectiveness. In Chapter 2 we describe existing data compression techniques to which our algorithms must be compared. We present research results in both of the component areas of data compression, modeling and coding. In Chapter 3 we present our solution to a specific data compression problem in which the model is fixed. Both use of memory and decode speed are constrained by the application. We design an implementation of the coding component that meets the

demands of the specific problem. Chapters 4 and 5 present work on general-purpose data compression algorithms that provide very good compression while having modest memory requirements and executing quickly. We employ a standard coding technique. Our contribution involves adapting a model to the demands of the application domain. Chapter 6 includes a summary of our contributions and suggestions for future research.

CHAPTER 1
Background Concepts

A simple characterization of data compression is that it involves transforming a string of characters in some representation (such as ASCII) into a new string (e.g., of bits) that contains the same information but whose length is as small as possible. There are two aspects to the process of data compression: modeling and coding. *Modeling* involves constructing a representation of the source that generates the data being compressed. Modeling addresses the question of how to partition the original data into basic units and what type of statistics, if any, to collect. *Coding* involves mapping the basic units into the compressed representation. The coding component takes information supplied by the modeler and translates this information into a sequence of bits. Designing a data compression algorithm entails selecting a modeling component and a coding component that work well together. The separation between modeling and coding is not always apparent in descriptions of data compression algorithms because the two features are not completely independent. Certain types of models correspond more naturally to certain types of codes and vice versa. The constraints of a particular application may also impact the connection between the choice of model and the choice of code.

The following section provides definitions of concepts necessary to a discussion and comparison of data compression methods. We use the string of characters $EXAMPLE =$ "*aa bbb cccc ddddd eeeeee fffffffgggggggg*" to illustrate the concepts defined. We talk about compressing a *string* of characters rather than a file or a stream. We frame our discussion in terms of compressing data to be stored or transmitted interchangeably.

1. Definitions

A *model* is a representation of the source that generates the data being compressed. The model partitions the input string into basic units called *source messages* which can be thought of as words over a source alphabet α. These basic units may be single symbols from the source alphabet, or they may be sequences of symbols. A *code* is a mapping of source messages into *codewords* (words over the code alphabet β). For string *EXAMPLE*, the source alphabet $\alpha = \{a, b, c, d, e, f, g, \ space\}$. For purposes of explanation, β will be

taken to be $\{0,1\}$. When source messages of variable length are allowed, the questions of how these basic units are selected and how a message *ensemble* (sequence of messages) is parsed into individual messages arise. *Defined-word models* are models in which the basic units are defined, but not necessarily fixed, prior to code construction. For example, in text file processing each character or each word may constitute a message. In *free-parse* models, the compression algorithm defines the set of source messages dynamically as it parses the ensemble. These are different from defined-word models in that there is no rule governing the selection of source messages (i.e., any string over the input alphabet is eligible for selection as a basic unit of compression). When the set of basic units permits multiple ways of partitioning a string (e.g., given basic units *a* and *aa*, there are several ways to partition the string *aaaaa*), a parsing strategy must be defined as part of the compression model. A common parsing strategy is the *greedy* method in which the longest source message matching a prefix of the input string is selected for coding at each step of the algorithm.

A code is *distinct* if the mapping from source messages to codewords is one-to-one, and a distinct code is *uniquely decodable* if every codeword is identifiable when immersed in a sequence of codewords. A uniquely decodable code is a *prefix code* if and only if no codeword is a proper prefix of any other codeword. Prefix codes are *instantaneously decodable.* That is, they have the desirable property that the coded message can be parsed into codewords without the need for lookahead. A *minimal* prefix code is a prefix code such that, if x is a proper prefix of some codeword, then $x\sigma$ is either a codeword or a proper prefix of a codeword for each letter σ in β. The minimality constraint prevents the use of codewords that are longer than necessary.

The process of transforming a source ensemble (or input string) into a coded message is *coding* or *encoding.* The encoded message may be referred to as an *encoding* of the source ensemble. The algorithm that constructs the mapping and uses it to transform the source ensemble is called the *encoder* or *compressor.* The *decoder* or *decompressor* performs the inverse operation, restoring the coded message to its original form. The term *encode* (likewise *decode*) serves to conceal the role of the model in the data compression process, however the term is meant to encompass the entire compression (decompression) operation.

A data compression algorithm can be classified as either static or dynamic, according to whether its modeling component is static or dynamic. A model is static if the information of which it consists is fixed over the course of the coding process. In a dynamic model,

source message	probability	codeword
a	2/40	1001
b	3/40	1000
c	4/40	011
d	5/40	010
e	6/40	111
f	7/40	110
g	8/40	00
space	5/40	101

Figure 1
A Huffman code for *EXAMPLE* (code length=117)

the set of basic units may change over time, or the basic units may remain fixed while statistics such as frequency of occurrence of each source message are updated. We also classify the coding component as either static or dynamic. A static algorithm will employ a static code as well as a static model. The fact that the model is fixed means that there is no reason for the mapping to change. In a dynamic algorithm the model is changing, but this may or may not necessitate changes by the coder. For example, an algorithm that employs a dynamic model and maps the information provided by the model using Huffman coding requires that the Huffman code mapping be updated as the statistics of the model are updated. On the other hand, if the model simply provides information as to whether the source message being encoded is the most frequent, second most frequent, etc., and the coder maps "most frequent" to 1, "second most frequent" to 01, etc., then the code is static while the model is dynamic.

The classic static algorithm is Huffman coding based on individual characters [H52]. In Huffman coding, the assignment of codewords to source messages is based on the probabilities with which the source messages appear in the input. Messages that appear frequently are represented by short codewords and messages with smaller probabilities map to longer codewords. A Huffman code for the ensemble *EXAMPLE* is given in Figure 1. If *EXAMPLE* were coded using this Huffman mapping, the coded message would contain 117 bits.

source message	probability	codeword
a	2/6	10
b	3/6	0
space	1/6	11

Figure 2
A dynamic Huffman code for prefix "*aa bbb*" of *EXAMPLE*

In dynamic Huffman coding of characters, the model computes an approximation to the probabilities of occurrence "on the fly", as the ensemble is being encoded. The assignment of codewords to messages is based on the values of the relative frequencies of occurrence at each point in time. A message x may be represented by a short codeword early in the encoding process because it occurs frequently at the beginning of the ensemble, even though its probability of occurrence over the total ensemble is low. Later, when the more probable messages begin to occur with higher frequency, the short codeword will be mapped to one of the higher probability messages, and x will be mapped to a longer codeword. Figure 2 presents a dynamic Huffman code corresponding to the prefix "*aa bbb*" of *EXAMPLE*. Although the frequency of *space* over the entire message is greater than that of *b*, at this point in time *b* has higher frequency and therefore is mapped to the shorter codeword.

Dynamic Huffman coding of characters is just one of many dynamic data compression schemes. Dynamic methods are also referred to as *adaptive*, in that they adapt to changes in ensemble characteristics over time. We prefer the term adaptive because the fact that these codes adapt to changing characteristics is the source of their appeal.

Adaptive methods require only a single pass over the string being compressed. Static Huffman coding requires two passes: one pass to compute probabilities and determine the mapping, and a second pass for transmission. The mapping determined in the first pass of a static coding scheme must be transmitted by the encoder to the decoder. In one-pass methods the encoder defines and redefines the mapping dynamically during transmission. The decoder must define and redefine the mapping in sympathy, in essence learning the mapping as codewords are received.

2. Measuring Performance

In addition to compression performance, speed and memory requirements may be important criteria in the selection of a data compression algorithm. Encode speed and decode speed may be the same, or they may be different. Similarly, the memory required to decode may be different from that required to encode. The application may also place more importance on some of these measures than others.

When data is compressed, the goal is to reduce redundancy, leaving only information content. The definitions of information content and redundancy measure the effectiveness of a code, given a model. That is, they are properties of the source *assuming* a particular model of that source. The most common assumption is that the source ensemble is partitioned into source messages $a_1, \ldots, a_n$ that occur with fixed probabilities $p(a_1), \ldots, p(a_n)$. The measure of information of a source message a_i (in bits) is $-\lg p(a_i)$*. This definition has intuitive appeal. In the case that $p(a_i) = 1$, it is clear that a_i is not at all informative since it had to occur. Similarly, the smaller the value of $p(a_i)$, the more unlikely a_i is to appear, and the more we learn when it does.

The average information content over the source alphabet can be computed by weighting the information content of each letter by its probability of occurrence, yielding $\sum_{i=1}^{n}[-p(a_i)\lg p(a_i)]$. This quantity is referred to as the *entropy* of the source, and is denoted by H. Because the length of a codeword for message a_i must be sufficient to carry the information content of a_i, entropy imposes a lower bound on the number of bits required for the coded message. The total number of bits must be at least as large as the product of H and the length of the source ensemble. Given that message *EXAMPLE* is to be encoded one letter at a time, the entropy of its source can be calculated to be $H = 2.894$, using the probabilities given in Figure 1. Thus the minimum number of bits for any encoding of *EXAMPLE based on this model* is 116. The Huffman code given in Figure 1 does not quite achieve the theoretical minimum in this case.

Average codeword length and redundancy are defined for static codes to compare the performance of the code to the theoretical minimum [H52, SW49]. *Redundancy* is the difference between average codeword length and average information content. A more useful definition of redundancy, applicable to a wider variety of data compression systems, is that it is the difference between the compression achieved and that predicted by an entropy calculation *based on the model employed.* The entropy computation for a static

* lg denotes the base 2 logarithm

model is usually straightforward. For an adaptive model, however, there may be no obvious way to compute entropy. The amount of compression yielded by a coding scheme may also be measured by a *compression ratio*. We define compression ratio to be (size of compressed representation)/(size of original string). We may express the ratio as the percentage of the input file size remaining after compression. We note that this definition is applicable to any data compression system since it makes no assumptions about either the modeling or the coding component of the algorithm. The measure depends only on file sizes before and after compressing.

CHAPTER 2
The Data Compression Landscape

In this chapter we present a portrait of the array of data compression algorithms currently in use. Designing a data compression system entails selecting a modeling component and a coding component. In Sections 1 and 2 we describe the components (codes and models) used in state-of-the-art systems and in Section 3 the systems are described. In Chapters 4 and 5 we present comparisons of our algorithms with the techniques we describe here.

1. Codes

Codes may be either static or adaptive. A static model demands a static code while an adaptive model may be used in conjunction with either a static code or an adaptive one. Many, although not all, static codes have adaptive equivalents.

The classic static code was developed nearly 40 years ago in Huffman's well-known paper on minimum-redundancy coding [H52]. Huffman's algorithm provided the first solution to the problem of constructing minimum-redundancy codes. It is widely believed that Huffman coding is guaranteed to achieve the best possible compression ratio. There are several problems with this statement. The first is that it ignores the issue of modeling. Huffman coding alone does not even specify a data compression system. Secondly, Huffman coding is optimal only among codes with the property that each source message must be coded in an integral number of bits. While this may seem to be required of any code, in fact it is not. Arithmetic coding, a more recent development, dispenses with the restriction that each source message translates into an integral number of bits and, as a result, performs at least as well and often better than Huffman coding.

We describe Huffman coding in Section 1.1. In Section 1.2, codes that map the integers onto binary codewords are discussed. Since any finite alphabet may be enumerated, this type of code has general-purpose utility and could be substituted for Huffman coding in any compression algorithm. Our interest in these codes, however, derives from the fact that they have been used in connection with specific adaptive modeling techniques. This connection is discussed in Section 3.

model is usually straightforward. For an adaptive model, however, there may be no obvious way to compute entropy. The amount of compression yielded by a coding scheme may also be measured by a *compression ratio*. We define compression ratio to be (size of compressed representation)/(size of original string). We may express the ratio as the percentage of the input file size remaining after compression. We note that this definition is applicable to any data compression system since it makes no assumptions about either the modeling or the coding component of the algorithm. The measure depends only on file sizes before and after compressing.

CHAPTER 2
The Data Compression Landscape

In this chapter we present a portrait of the array of data compression algorithms currently in use. Designing a data compression system entails selecting a modeling component and a coding component. In Sections 1 and 2 we describe the components (codes and models) used in state-of-the-art systems and in Section 3 the systems are described. In Chapters 4 and 5 we present comparisons of our algorithms with the techniques we describe here.

1. Codes

Codes may be either static or adaptive. A static model demands a static code while an adaptive model may be used in conjunction with either a static code or an adaptive one. Many, although not all, static codes have adaptive equivalents.

The classic static code was developed nearly 40 years ago in Huffman's well-known paper on minimum-redundancy coding [H52]. Huffman's algorithm provided the first solution to the problem of constructing minimum-redundancy codes. It is widely believed that Huffman coding is guaranteed to achieve the best possible compression ratio. There are several problems with this statement. The first is that it ignores the issue of modeling. Huffman coding alone does not even specify a data compression system. Secondly, Huffman coding is optimal only among codes with the property that each source message must be coded in an integral number of bits. While this may seem to be required of any code, in fact it is not. Arithmetic coding, a more recent development, dispenses with the restriction that each source message translates into an integral number of bits and, as a result, performs at least as well and often better than Huffman coding.

We describe Huffman coding in Section 1.1. In Section 1.2, codes that map the integers onto binary codewords are discussed. Since any finite alphabet may be enumerated, this type of code has general-purpose utility and could be substituted for Huffman coding in any compression algorithm. Our interest in these codes, however, derives from the fact that they have been used in connection with specific adaptive modeling techniques. This connection is discussed in Section 3.

Arithmetic coding, presented in Section 1.3, takes a significantly different approach to data compression from that of the other coding methods. It does not construct a code, in the sense of a mapping from source messages to codewords. Arithmetic coding is capable of achieving compression results that are arbitrarily close to the entropy of the source.

1.1. Huffman Coding

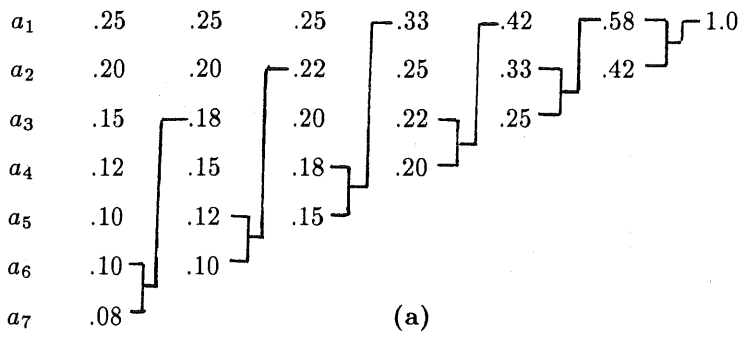
Huffman coding may be used either statically or adaptively. We describe first the original (static) Huffman coding algorithm. Huffman's algorithm, expressed graphically, takes as input a list of nonnegative weights $\{w_1, \ldots, w_n\}$ and constructs a full binary tree* whose leaves are labeled with these weights (the weights represent the probabilities of the source letters). Initially there is a set of singleton trees, one for each weight in the list. At each step in the algorithm the trees corresponding to the two smallest weights, w_i and w_j, are merged into a new tree whose weight is $w_i + w_j$ and whose root has two children that are the subtrees represented by w_i and w_j. The weights w_i and w_j are removed from the list and $w_i + w_j$ is inserted into the list. This process continues until the weight list contains a single value. If, at any time, there is more than one way to choose a smallest pair of weights, any such pair may be chosen. The Huffman algorithm is demonstrated in Figure 3.

Huffman's algorithm determines the lengths of the codewords to be mapped to each of the source letters a_i. There are many alternatives for specifying the actual digits so as to obtain a prefix code. The usual assignment entails labeling the edge from each parent to its left child with the digit 0 and the edge to the right child with 1. The codeword for each source letter is the sequence of labels along the path from the root to the leaf node representing that letter. The codewords for the source of Figure 3, in order of decreasing probability, are $\{01, 11, 001, 100, 101, 0000, 0001\}$. Huffman's algorithm is guaranteed to produce a minimum redundancy code [H52]. Gallager has proved an upper bound on the redundancy of a Huffman code of approximately $p_n + 0.086$, where p_n is the probability of the least likely source message [G78].

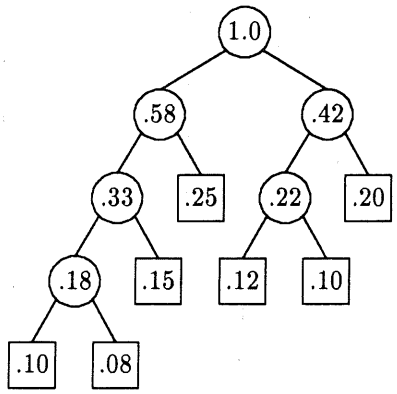
The code resulting from the above algorithm is, by nature, static. Adaptive Huffman coding is supplied by an adaptive model with a running estimate of source message probabilities. The code must be adapted so as to remain optimal for the current estimates. That is, the tree must be modified to reflect changes in the statistics supplied by the model.

---

* a binary tree is full if every node has either zero or two children



(a)



(b)

Figure 3

The Huffman process (a) the list (b) the tree

Adaptive Huffman coding becomes the problem of efficiently maintaining a binary tree rather than the static problem of building a tree.

Adaptive Huffman coding was first conceived independently by Faller and Gallager [F73, G78]. Knuth contributed improvements to the original algorithm [K85] and the resulting algorithm is referred to as algorithm FGK. A more recent version of adaptive Huffman coding (algorithm Λ) is described by Vitter [V87]. Vitter proves that neither algorithm FGK nor algorithm Λ can perform substantially worse than static Huffman coding. In practice the adaptive codes provide better compression than their static equivalents and require only a single pass over the data.

1.2. Fixed Codes

Most fixed codes are *universal*; that is, they map source messages to codewords so that the resulting average codeword length is bounded by $c_1 H + c_2$. The potential compression offered by a universal code clearly depends on the magnitudes of the constants c_1 and c_2. If $c_1 = 1$, the code has average codeword length that approaches the theoretical minimum and is defined as being *asymptotically optimal.* The use of fixed codes simplifies the encoding and decoding processes. The algorithm needs only to rank the source messages in order of decreasing frequency (exact frequency values are not needed) and map the pre-determined codewords to them in order of increasing codeword length.

We may think of a universal code as representing an enumeration of the source messages, or as representing the integers, which provide an enumeration. Elias defines a sequence of universal coding schemes that map the set of positive integers onto the set of binary codewords [E75]. The first Elias code is one that is simple but not optimal. This code, γ, maps an integer x onto the binary value of x prefaced by $\lfloor \lg x \rfloor$ zeros. The second code, δ, maps an integer x to a codeword consisting of $\gamma(1 + \lfloor \lg x \rfloor)$ followed by the binary value of x with the leading 1 deleted. The resulting codeword has length $\lfloor \lg x \rfloor + 2\lfloor \lg(1 + \lfloor \lg x \rfloor) \rfloor + 1$. This concept can be applied recursively to shorten the codeword lengths, but the benefits decrease rapidly. The code δ is asymptotically optimal.

A second sequence of universal coding schemes, based on the Fibonacci numbers, is defined by Apostolico and Fraenkel [AF87]. While the Fibonacci codes are not asymptotically optimal, they compare well to the Elias codes as long as the number of source messages is not too large. Fibonacci codes have the additional attribute of robustness, which manifests itself by the local containment of errors. This aspect of Fibonacci codes, and the robustness of other codes and data compression techniques, is discussed in the survey by Lelewer and Hirschberg [LH87].

Yet another family of universal codes is defined by Fiala and Green to be used as part of an improved Ziv-Lempel compression algorithm [FG89]. These codes are called *start-step-stop* codes and are parameterized by the three values start, step, and stop. A start-step-stop code yields k sets of binary codewords where $k = (stop - start)/step + 1$. Each codeword in the j^{th} set has a prefix consisting of $j - 1$ ones followed by a zero (for $1 \leq j < k$; for $j = k$ the 0 can be omitted). The suffixes of the codewords are binary strings of length $start + (j - 1) * step$. For example, the start-step-stop code (3,2,9) contains four sets of codewords of the following form: 0xxx, 10xxxxx,110xxxxxxx,111xxxxxxxxx; for a

source message	probability	cumulative probability	range
A	.2	.2	$[0, .2)$
B	.4	.6	$[.2, .6)$
C	.1	.7	$[.6, .7)$
D	.2	.9	$[.7, .9)$
#	.1	1.0	$[.9, 1.0)$

Figure 4
Arithmetic coding

total of 680 different codewords. The Elias code γ is equivalent to the start-step-stop code $(0,1,\infty)$ if the roles of 0 and 1 in the start-step-stop prefix are reversed.

1.3. Arithmetic Coding

In arithmetic coding a source ensemble is represented by an interval between 0 and 1 on the real number line. Each symbol of the ensemble narrows this interval. As the interval becomes smaller, the number of bits needed to specify it grows. Arithmetic coding assumes a probabilistic model of the source and uses the probabilities of the source messages to successively narrow the interval used to represent the ensemble. A high probability message narrows the interval less than a low probability message, so that high probability messages contribute fewer bits to the coded ensemble. The method begins with an unordered list of source messages and their probabilities. The number line is partitioned into subintervals based on cumulative probabilities.

A small example will be used to illustrate the idea of arithmetic coding. Given source messages $\{A, B, C, D, \#\}$ with probabilities .2, .4, .1, .2, and .1, Figure 4 demonstrates the initial partitioning of the number line. When encoding begins, the source ensemble is represented by the entire interval $[0, 1)$. For the ensemble $AADB\#$, the first A reduces the interval to $[0, .2)$ and the second A to $[0, .04)$ (the first $\frac{1}{5}$ of the previous interval). The D further narrows the interval to $[.028, .036)$ ($\frac{1}{5}$ of the previous size, beginning 70% of the distance from left to right). The B narrows the interval to $[.0296, .0328)$, and the # yields a final interval of $[.03248, .0328)$. The interval, or alternatively any number i within the interval, may now be used to represent the source ensemble.

source message	probability	cumulative probability	range
a	.05	.05	[0, .05)
b	.075	.125	[.05, .125)
c	.1	.225	[.125, .225)
d	.125	.35	[.225, .35)
e	.15	.5	[.35, .5)
f	.175	.675	[.5, .675)
g	.2	.875	[.675, .875)
space	.125	1.0	[.875, 1.0)

Figure 5
The arithmetic code for *EXAMPLE*

The size of the final subinterval determines the number of bits needed to specify a number in that range. The number of bits needed to specify a subinterval of $[0, 1)$ of size s is $-\lg s$. Since the size of the final subinterval is the product of the probabilities of the source messages in the ensemble (that is, $s = \prod_{i=1}^{N} p(\text{source message } i)$ where N is the length of the ensemble), we have $-\lg s = -\sum_{i=1}^{N} \lg p(\text{source message } i) = -\sum_{i=1}^{n} p(a_i) \lg p(a_i)$, where n is the number of unique source messages $a_1, a_2, \cdots a_n$. Thus, the number of bits generated by the arithmetic coding technique is exactly equal to entropy, H.

In order to recover the original ensemble, the decoder must know the model of the source used by the encoder (e.g., the source messages and associated ranges) and a single number within the interval determined by the encoder. Decoding consists of a series of comparisons of the number i to the ranges representing the source messages. For the example of Figure 4, i might be .0325. The decoder uses i to simulate the actions of the encoder. Since i lies between 0 and .2, he deduces that the first letter was A (since the range $[0, .2)$ corresponds to source message A). This narrows the interval to $[0, .2)$. The decoder can now deduce that the next message will further narrow the interval in one of the following ways: to $[0, .04)$ for A, to $[.04, .12)$ for B, to $[.12, .14)$ for C, to $[.14, .18)$ for D, and to $[.18, .2)$ for $\#$. Since i falls into the interval $[0, .04)$, he knows that the second message is again A. This process continues until the entire ensemble has been recovered.

The arithmetic code based on a character-by-character model of the string *EXAMPLE* is given in Figure 5. The final interval size is: $p^2(a) * p^3(b) * p^4(c) * p^5(d) * p^6(e) *$

source message	probability	cumulative probability	range
A	.2	.2	$[0, .2)$
B	.4	.6	$[.2, .6)$
C	.1	.7	$[.6, .7)$
D	.2	.9	$[.7, .9)$
#	.1	1.0	$[.9, 1.0)$

Figure 4
Arithmetic coding

total of 680 different codewords. The Elias code γ is equivalent to the start-step-stop code $(0,1,\infty)$ if the roles of 0 and 1 in the start-step-stop prefix are reversed.

1.3. Arithmetic Coding

In arithmetic coding a source ensemble is represented by an interval between 0 and 1 on the real number line. Each symbol of the ensemble narrows this interval. As the interval becomes smaller, the number of bits needed to specify it grows. Arithmetic coding assumes a probabilistic model of the source and uses the probabilities of the source messages to successively narrow the interval used to represent the ensemble. A high probability message narrows the interval less than a low probability message, so that high probability messages contribute fewer bits to the coded ensemble. The method begins with an unordered list of source messages and their probabilities. The number line is partitioned into subintervals based on cumulative probabilities.

A small example will be used to illustrate the idea of arithmetic coding. Given source messages $\{A, B, C, D, \#\}$ with probabilities .2, .4, .1, .2, and .1, Figure 4 demonstrates the initial partitioning of the number line. When encoding begins, the source ensemble is represented by the entire interval $[0, 1)$. For the ensemble $AADB\#$, the first A reduces the interval to $[0, .2)$ and the second A to $[0, .04)$ (the first $\frac{1}{5}$ of the previous interval). The D further narrows the interval to $[.028, .036)$ ($\frac{1}{5}$ of the previous size, beginning 70% of the distance from left to right). The B narrows the interval to $[.0296, .0328)$, and the # yields a final interval of $[.03248, .0328)$. The interval, or alternatively any number i within the interval, may now be used to represent the source ensemble.

source message	probability	cumulative probability	range
a	.05	.05	[0, .05)
b	.075	.125	[.05, .125)
c	.1	.225	[.125, .225)
d	.125	.35	[.225, .35)
e	.15	.5	[.35, .5)
f	.175	.675	[.5, .675)
g	.2	.875	[.675, .875)
space	.125	1.0	[.875, 1.0)

Figure 5
The arithmetic code for *EXAMPLE*

The size of the final subinterval determines the number of bits needed to specify a number in that range. The number of bits needed to specify a subinterval of $[0, 1)$ of size s is $-\lg s$. Since the size of the final subinterval is the product of the probabilities of the source messages in the ensemble (that is, $s = \prod_{i=1}^{N} p(\text{source message } i)$ where N is the length of the ensemble), we have $-\lg s = -\sum_{i=1}^{N} \lg p(\text{source message } i) = -\sum_{i=1}^{n} p(a_i) \lg p(a_i)$, where n is the number of unique source messages $a_1, a_2, \cdots a_n$. Thus, the number of bits generated by the arithmetic coding technique is exactly equal to entropy, H.

In order to recover the original ensemble, the decoder must know the model of the source used by the encoder (e.g., the source messages and associated ranges) and a single number within the interval determined by the encoder. Decoding consists of a series of comparisons of the number i to the ranges representing the source messages. For the example of Figure 4, i might be .0325. The decoder uses i to simulate the actions of the encoder. Since i lies between 0 and .2, he deduces that the first letter was A (since the range $[0, .2)$ corresponds to source message A). This narrows the interval to $[0, .2)$. The decoder can now deduce that the next message will further narrow the interval in one of the following ways: to $[0, .04)$ for A, to $[.04, .12)$ for B, to $[.12, .14)$ for C, to $[.14, .18)$ for D, and to $[.18, .2)$ for $\#$. Since i falls into the interval $[0, .04)$, he knows that the second message is again A. This process continues until the entire ensemble has been recovered.

The arithmetic code based on a character-by-character model of the string *EXAMPLE* is given in Figure 5. The final interval size is: $p^2(a) * p^3(b) * p^4(c) * p^5(d) * p^6(e) *$

$p^7(f) * p^8(g) * p^5(space)$. The number of bits needed to specify a value in the interval is $-\lg(1.44*10^{-35}) = 115.7$. So excluding overhead, arithmetic coding transmits *EXAMPLE* in 116 bits, one less bit than static Huffman coding.

Witten et al. provide an implementation of arithmetic coding that separates the model of the source from the coding process [WNC87]. The model is in a separate program module and is consulted by the encoder and the decoder at every step in the processing. This implementation delineates clearly the separation between modeling and coding in the data compression process.

Adaptive Huffman coding is very different from static Huffman coding because the former involves building a code tree while the latter requires maintaining a dynamic tree. The arithmetic coding method is based on the frequencies and cumulative frequencies of the basic units being coded and there is no data structure dependence here. Thus no modifications are necessary to produce an adaptive version of arithmetic coding. In a simple implementation of arithmetic coding the frequency values are stored in an array in decreasing order and the cumulative frequency values are stored in a second array. Maintaining the frequency information involves incrementing the appropriate frequency and cumulative frequency values and reordering the arrays as necessary. An index is used to map source messages to positions in the frequency arrays and vice versa so that when a message moves up in the frequency array its index value is changed to reflect the move.

2. Models

Most models for data compression are probabilistic in nature. That is, they consist of a set of source messages and associated probabilities or frequencies. Probabilistic models are combined with probabilistic codes such as Huffman coding or arithmetic coding to form a data compression system. The simplest, and most obvious, models are those that represent the input file as a sequence of characters and code each character using either a fixed-length code (e.g., ASCII code) or a probabilistic code based on frequency of occurrence. The probabilistic code may be either static or adaptive, depending on whether the modeler computes probabilities in a preliminary pass over the entire input or on the fly.

Most static models have adaptive equivalents and the adaptive counterparts generally provide more effective compression. In fact, Bell et al. prove that over a large range of circumstances there is an adaptive model that will be only slightly worse than *any*

static model, while a static model can be arbitrarily worse than an adaptive counterpart [BCW90]. All of the newer data compression models are adaptive. In this section we present

(1) the dictionary model used by many Ziv-Lempel data compression systems,

(2) the move-to-front model, which uses self-organizing lists to exploit locality, and

(3) finite-context modeling, an important and relatively recent approach.

2.1. Dictionary Models

In *dictionary modeling* compression is achieved by replacing groups of consecutive characters with indices into a dictionary. A dictionary model may be static, but is more often adaptive with the algorithm constructing the dictionary as it encodes the source ensemble. A typical dictionary model consists of a rule for parsing the input into substrings of bounded length. Compression is achieved when a long string of source symbols is replaced by a single codeword. This strategy is effective at exploiting redundancy due to symbol frequency, character repetition, and high-usage patterns [W84]. The Ziv-Lempel family of compression algorithms employs dictionary models. In Section 3.1 we describe two of the more promising algorithms in this large and growing family. More complete descriptions of the Ziv-Lempel family (and dictionary models) can be found in the books by Storer and Bell et al. [ST88, BCW90].

2.2. The Move-to-Front Model

The move-to-front model is a defined-word model that uses a move-to-front data structure to maintain the list of source messages. The move-to-front list organizing strategy is inherently dynamic, so the move-to-front data compression model is adaptive with no natural static equivalent. As in any adaptive model, encoder and decoder maintain identical representations of the data structure, in this case message lists that are updated at each transmission using the move-to-front heuristic. When message m occurs in the input and m is on the encoder's list, the encoder transmits m's current position. The encoder then updates its list by moving m to position 1 and shifting each of the other messages down one position. The decoder similarly alters its list. If m is being transmitted for the first time, then $k+1$ is the "position" transmitted, where k is the number of distinct messages transmitted so far. Some representation of the message itself must be transmitted as well, but just this first time. Again, m is moved to position 1 by both encoder and decoder

subsequent to its transmission. If encoder and decoder maintain lists of single characters, the ensemble "*abcadeabfd*" is modeled as: 1 *a* 2 *b* 3 *c* 3 4 *d* 5 *e* 3 5 6 *f* 5.

As the example shows, the move-to-front model transmits each source message once. The rest of its transmission consists of encodings of list positions. The strategy inherent in the use of the self-organizing list is to exploit locality of reference, the tendency for source messages to occur frequently for short periods of time then fall into long periods of disuse. The move-to-front model was developed independently by Bentley et al., Elias, and Ryabko [BSTW86, E87, R87].

2.3. Finite-Context Models

Context modeling has emerged as the most promising new approach to compressing text. A *finite-context model* predicts successive characters taking into account the context provided by characters already seen. What is meant by *predict* here is that previous characters are used in determining the number of bits used to encode the current character. The idea of a context consisting of a few previous characters is very reasonable when the data being compressed is natural language. We all know that the character following *q* is all but guaranteed to be *u* and that given the context *now is the time for all good men to come to the aid of*, the phrase *their country* is bound to follow. Although the technique of context modeling was developed and is clearly appropriate for compressing natural language, context models provide very good compression over a wide range of file types.

A model that uses i previous characters to predict the current character is referred to as an *order-i context model.* When $i = 0$, no context is used and the text is simply coded one character at a time. When $i = 1$, the previous character is used in encoding the current character; when $i = 2$, the previous two characters are used, and so on. Context modeling may be employed statically but is more often used adaptively. Combining the finite-context model with a probabilistic code exploits the context information. Context modeling will be discussed in more detail in Chapters 4 and 5.

Another promising new model is the *finite-state model*, or *Markov model*, based on a finite-state machine. The finite-state model can represent context information by using a state for each context. The finite-state model is potentially more powerful than the finite-context model because it can represent additional characteristics of the input, such as "every fourth character in the file is a zero" or "every sequence of *a*'s has even length".

Horspool and Cormack describe an adaptive finite-state model DMC (for dynamic Markov compression) [HC86, CH87]. Due to the way in which states are added to the model, however, DMC does not attain the potential power of finite-state models. Bell and Moffat show that the DMC model is equivalent to a finite-context model [BM89]. Further research is required to determine effective ways of building finite-state machines that model the source of a message ensemble.

3. Systems

The state of the art in practical data compression is represented by the UNIX* utility *compress*. In addition to the UNIX utility, personal-computer versions of *compress* have been in wide use for a number of years. Newer algorithms, including the Ziv-Lempel-based algorithm FG, and the context-model-based algorithm PPMC, provide dramatically better compression performance than *compress*. Each of these methods has advantages and disadvantages in terms of compression performance, use of memory, and speed of compression and decompression. We describe the Ziv-Lempel-based systems *compress* and algorithm FG in Section 3.1. In Section 3.2, we present *compact*, another UNIX utility, and in Section 3.3 algorithm BSTW, a move-to-front method. Algorithm PPMC is described in Section 3.4.

3.1. Compress and Algorithm FG

Compress and algorithm FG employ dictionary models as defined by Ziv and Lempel and described in Section 2.1 [ZL78]. *Compress* parses the source ensemble into a collection of segments of gradually increasing length. At each encoding step, the longest prefix of the source ensemble that matches an existing dictionary entry (α) is parsed off, along with the character (c) following this prefix in the ensemble. The new source message, αc, is added to the dictionary. The encoder transmits the pair (i, c) where i is the pointer to the existing entry and c is the appended character. For ensemble *EXAMPLE compress* will build a dictionary consisting of: { a, $a_{\sqcup}$, b, bb, $_{\sqcup}$, c, cc, $c_{\sqcup}$, d, dd, $dd_{\sqcup}$, e, ee, eee, $_{\sqcup}f$, f, ff, fff, g, gg, ggg } (where $_{\sqcup}$ is used to represent *space*).

Compress uses fixed-length codewords. Ziv-Lempel coding is asymptotically optimal, meaning that the redundancy approaches zero as the length of the source ensemble tends to infinity. It should be clear, however, that *compress* tends to be quite inefficient during the initial portion of the message ensemble. For example, assuming 3-bit codewords

* UNIX is a trademark of AT&T Bell Laboratories.

for characters *a* through *g* and *space* and 5-bit codewords for pointers, the Ziv-Lempel algorithm transmits 173 bits for ensemble *EXAMPLE*. This compares poorly with the Huffman and arithmetic coding methods as shown in Figures 1 and 5. The ensemble must be sufficiently long for the procedure to build up enough symbol frequency experience to achieve good compression.

If the size of the dictionary is not sufficiently large, Ziv-Lempel codes may also rise slowly to reasonable efficiency, maintain good performance briefly, and fail to make any gains once the table is full and messages can no longer be added. *Compress* uses a large dictionary, and clears the table and starts from scratch when it becomes full. This occurs infrequently, however, since *compress* also monitors the compression ratio as it executes and clears the dictionary when the ratio begins to deteriorate.

Algorithm FG has the advantages of improved compression and improved utilization of runtime memory over *compress*. *Compress* uses 450 Kbytes of memory and provides compression in the range of 45–50 percent. Algorithm FG's compression performance is about 30 percent better than that of *compress* and it requires just 186 Kbytes for encoding and 130 Kbytes for decoding. These improvements are achieved at the expense of speed, however. Algorithm FG encodes text at an average rate of 6,000 characters per second (cps) and decodes approximately 11,000 cps while *compress* encodes and decodes at about 15,000 cps [BCW90, Mo89]. The improvements in algorithm FG come from the data structure, a Patricia trie, and the use of variable-length coding of the integers [FG89]. The codes used are the start-step-stop codes described in Section 1.2 of this chapter.

3.2. Compact

The UNIX utility *compact* employs an adaptive model in which single characters and their frequencies are recorded. The model is supported by an adaptive Huffman code, algorithm FGK. *Compact* has modest memory requirements, but runs slowly (encoding and decoding about 1600 cps) and has relatively poor compression performance (leaving about 62% of the original file size). The primary reason for the poor compression performance is the simplicity of the model used in *compact*. All of the methods to which we compare it use much more sophisticated models. The coding component is responsible for the lack of speed. Maintaining dynamic Huffman trees is difficult to do quickly.

3.3. Algorithm BSTW

Algorithm BSTW uses the move-to-front model defined in Section 2.2 of this chapter, where the source messages maintained in the self-organizing data structure are words [BSTW86]. There are two codes used in conjunction with the move-to-front model, one to code list positions and another to code words when they occur for the first time. Bentley et al. use the Elias codes defined in Section 1.2 of this chapter to code list positions. Coding the words when they occur for the first time constitutes a data compression problem within a data compression problem. Both a model and a code are required. The implementation of algorithm BSTW for which we report compression results in Chapter 5 models the new words using frequencies of individual characters and codes the model using adaptive arithmetic coding. This implementation provides better compression than the state-of-the-art *compress*, but is not competitive with newer compression systems such as algorithm FG.

3.4. Algorithm PPMC

Algorithm PPMC employs an adaptive finite-context model and codes the context information using adaptive arithmetic coding. The model provides a frequency distribution for each context (each character in the order-1 case and each pair of characters in the order-2 case). Each frequency distribution forms the basis of an arithmetic code and these are used to map events into code bits. Huffman coding may be used in concert with finite-context models but will generally perform less effectively.

Algorithm PPMC will be described in more detail in Chapter 4 when we contrast it with our finite-context-based algorithm. Algorithm PPMC provides very good compression (approximately 30 percent on average), but has a large memory requirement (500 Kbytes) and executes slowly (encoding and decoding approximately 2000 cps).

CHAPTER 3
Efficient Decoding of Prefix Codes

In this chapter a special case of the data compression problem is presented. The application involves a powerful encoder that transmits a compressed file to a decoder that has severely constrained memory. A data structure that achieves minimum storage is presented, and alternative methods that sacrifice a small amount of storage to attain faster decoding are described. In Section 1 we expand on the definition of the specific problem addressed in this chapter. In Section 2 we discuss previous work on similar problems and show that none of this previous work is applicable to our specific problem. We present four solutions to the problem, two based on one approach in Section 3, and two based on a different idea in Section 4. In Section 5 we discuss additional implementation considerations and Section 6 provides a summary of the chapter.

1. The Application

The work we describe in this chapter is based on a specific data compression application in which:

(a) textual data is to be transmitted and received over a communication line,

(b) decoding must be performed on-line, and

(c) the amount of memory available during the decode operation is very limited.

The encoder in our data compression system is allowed substantial computational resources. It can expend significant time and space to find a compact representation of the source text. Once the representation is constructed, it will be transmitted to the decoder. The decoder may be viewed as a special-purpose translator with very limited space. This space limitation provides an interesting challenge. While technology is providing increasing amounts of memory at low cost, minimizing the use of memory will always be a goal in mass production applications of data compression.

We employ a static dictionary compression technique, that is, an algorithm that compresses a source text by replacing strings of characters in the source by pointers to a

dictionary. The *dictionary* is a collection of n strings of varying lengths. Long dictionary entries have higher potential for compression than short ones in that we replace a large number of characters with a single codeword. However, we must also take into account the frequency with which a dictionary entry occurs in the source text. We want to assign short codewords to frequently-occurring strings. If a string occurs only rarely its codeword may be too long to provide good compression even though the string being replaced is itself quite long. The degree of compression to be achieved by a dictionary compression system is largely dependent on the choice of the dictionary. It is also necessary, however, to represent the pointers efficiently. We choose to represent pointers by prefix codes based on the relative frequencies of the dictionary entries they represent. The Huffman code is the most widely-known prefix code and is minimal in that it provides the best compression of any prefix code applied to a fixed dictionary [H52]. Arithmetic codes, which are not prefix codes, can provide better compression than the Huffman code when applied to the same dictionary [WNC87]. This improved compression is possible because arithmetic codes are not constrained to map an integer number of bits to each dictionary entry.

An offsetting advantage of Huffman codes is that they are more robust. While an error in a single bit will prevent the bits that follow from being correctly decoded by an arithmetic decoder, Huffman codes tend to resynchronize quickly, thus localizing damage [LH87]. A more important consideration in terms of the present application is the fact that arithmetic coding uses the frequencies of the dictionary entries during decoding. Our methods do not require the table of frequencies, and as a result we are able to decode with a much smaller space requirement. For these reasons we elect to use Huffman coding for our application.

The compressed version of the source text consists of a representation of:

(1) the encoding dictionary,

(2) its prefix code, and

(3) the sequence of codes that can be expanded to recover the original text.

Most of the compression is achieved by choosing an appropriate dictionary. The computation of the corresponding prefix code is straightforward. However, the method of representing the dictionary and the prefix code also affects the resulting compression ratio (for moderate-sized files, the representation choice can have a significant impact on the

symbol	storage requirement for	typical value
A	an address	2 bytes
C	number of characters in a dictionary entry	1 byte
N	an integer between 1 and $n+1$	2 bytes
M	number of codewords of a given length	1 byte
B	length of a codeword (in bits)	1 byte
V	value of a codeword	2 bytes
	meaning	
L	$max - min + 1$	13
max	length of longest codeword (in bits)	12–16
min	length of shortest codeword (in bits)	1–3

Table 1
Variables used to define storage requirements

compression ratio). The encoder in our application must construct a representation that is compact and that our space-limited decoder can translate efficiently. The way in which the encoder represents the dictionary and the prefix code is the focus of our work. We partition the encoding dictionary into two parts:

(1a) a stream of characters, and

(1b) information that permits parsing this stream into individual dictionary entries (e.g., the lengths of the entries or their starting positions).

All of our methods prepend the stream of characters to the encoded text and store the characters as part of the decode data structure. It is in the way that (1b) and (2) are represented that the methods differ. We will compare the decode space efficiencies of our methods and the amount of *representation overhead* they incur. We define representation overhead to be the number of bytes in the compressed text used to represent items (1b) and (2). We allow the decoder some limited set-up time to receive the code representation (items 1 and 2) and store the information needed for performing translation. Except for the delay due to set up, the decoder must operate on-line. That is, the time required for decoding must be proportional to the size of the expanded source.

string	frequency
abcd	10
rst	9
wxyz	15
qu	7
lm	2
ps	2
the	22

Figure 6
An example dictionary

In order that our methods may be presented in the most general form, we define the variables listed in Table 1. It should be noted that $N \geq \lg n$ bits, that $M \leq V$, that $B \leq V$, and that $A \geq C$ since we must be able to access any dictionary entry with an address. Figure 6 presents a small example dictionary, which we use to illustrate our methods.

2. Previous Methods

A number of papers have appeared on the subject of implementations of Huffman encoding and decoding. These implementations apply to any prefix code. The more recent of these papers, those by Sieminski and Choueka et al., concentrate on fast implementations and reduce processing time by avoiding manipulation of individual bits [Si88, CFKP86]. However, a price is paid for the reduced time requirements in the form of increased memory requirements. Sieminski's method requires 64 Kbytes to store the decode tables for a simple situation in which the dictionary contains only 127 individual characters. The size of the decode tables grows exponentially if dictionary entries longer than one character are used [Si88]. The method of Choueka et al. requires $O(n^2)$ extra space where n is the number of dictionary entries [CFKP86]. While processing time is of concern, our primary criterion is the efficient use of internal memory during decoding. Thus these methods are inappropriate for our purposes.

Hankamer describes a modified Huffman procedure with reduced memory requirements [H79]. The reduced memory requirements are attained by reducing the size of the

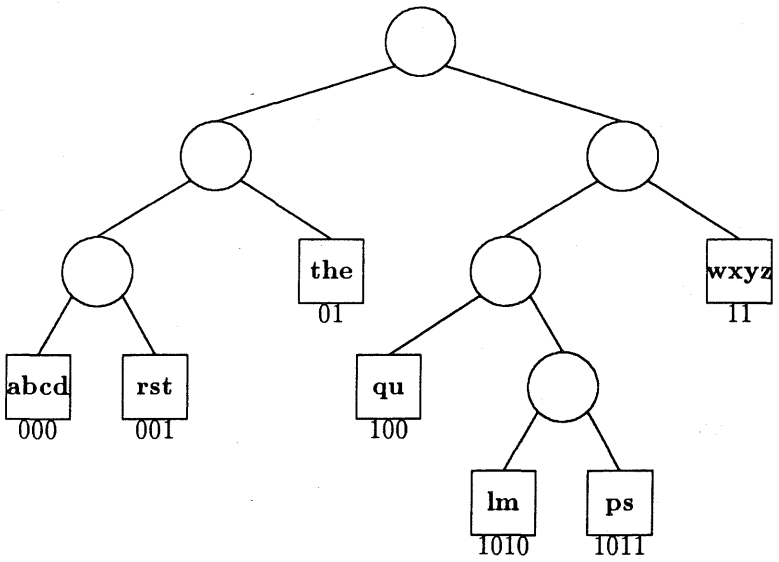


Figure 7
A Huffman tree for example dictionary

dictionary and computing a suboptimal Huffman code. Hankamer's method assumes fixed-length dictionary entries and there is no obvious extension to variable-length entries. This fact coupled with the loss of optimality renders the method inappropriate for our needs. Tanaka gives a finite automaton-based Huffman decoding algorithm [T87]. This method assumes single character dictionary entries. A straightforward modification to allow for variable-length entries is similar to our Method A1 (which follows) in terms of execution speed, but requires approximately 67 percent more memory.

3. Method A

Our first solution to the problem of decoding in restricted memory uses the Huffman code tree to represent the dictionary. However, we do not use the obvious linked implementation in which each internal node contains pointers to its left and right subtrees, as the space requirements of this representation are prohibitive. Instead, Method A1 employs an implicit representation of the tree structure. Method A2 is a variation of Method A1 that provides improved storage utilization.

string	frequency
abcd	10
rst	9
wxyz	15
qu	7
lm	2
ps	2
the	22

Figure 6
An example dictionary

In order that our methods may be presented in the most general form, we define the variables listed in Table 1. It should be noted that $N \geq \lg n$ bits, that $M \leq V$, that $B \leq V$, and that $A \geq C$ since we must be able to access any dictionary entry with an address. Figure 6 presents a small example dictionary, which we use to illustrate our methods.

2. Previous Methods

A number of papers have appeared on the subject of implementations of Huffman encoding and decoding. These implementations apply to any prefix code. The more recent of these papers, those by Sieminski and Choueka et al., concentrate on fast implementations and reduce processing time by avoiding manipulation of individual bits [Si88, CFKP86]. However, a price is paid for the reduced time requirements in the form of increased memory requirements. Sieminski's method requires 64 Kbytes to store the decode tables for a simple situation in which the dictionary contains only 127 individual characters. The size of the decode tables grows exponentially if dictionary entries longer than one character are used [Si88]. The method of Choueka et al. requires $O(n^2)$ extra space where n is the number of dictionary entries [CFKP86]. While processing time is of concern, our primary criterion is the efficient use of internal memory during decoding. Thus these methods are inappropriate for our purposes.

Hankamer describes a modified Huffman procedure with reduced memory requirements [H79]. The reduced memory requirements are attained by reducing the size of the

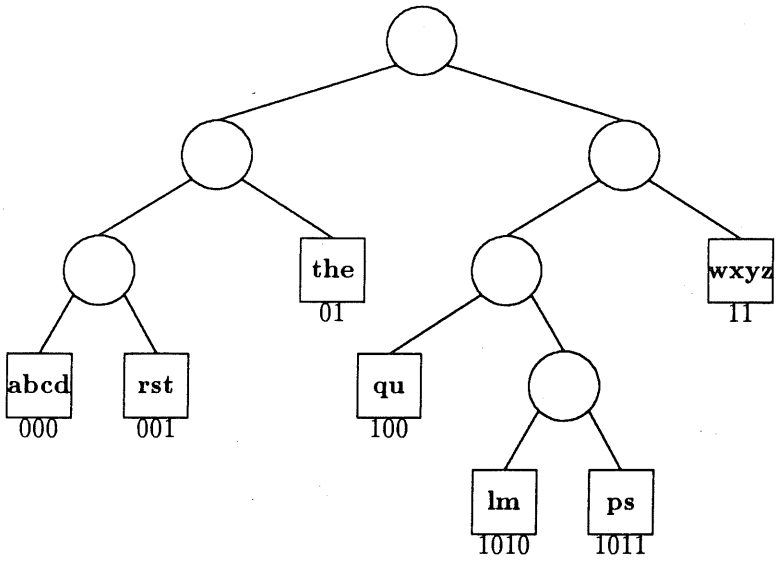


Figure 7
A Huffman tree for example dictionary

dictionary and computing a suboptimal Huffman code. Hankamer's method assumes fixed-length dictionary entries and there is no obvious extension to variable-length entries. This fact coupled with the loss of optimality renders the method inappropriate for our needs. Tanaka gives a finite automaton-based Huffman decoding algorithm [T87]. This method assumes single character dictionary entries. A straightforward modification to allow for variable-length entries is similar to our Method A1 (which follows) in terms of execution speed, but requires approximately 67 percent more memory.

3. Method A

Our first solution to the problem of decoding in restricted memory uses the Huffman code tree to represent the dictionary. However, we do not use the obvious linked implementation in which each internal node contains pointers to its left and right subtrees, as the space requirements of this representation are prohibitive. Instead, Method A1 employs an implicit representation of the tree structure. Method A2 is a variation of Method A1 that provides improved storage utilization.

address	1	3	5	7	12	16	20
contents	$(0,20)$	$(0,16)$	$(0,12)$	$(1,4,abcd)$	$(1,3,rst)$	$(1,3,the)$	$(0,35)$
address	22	24	27	29	32	35	
contents	$(0,27)$	$(1,2,qu)$	$(0,32)$	$(1,2,lm)$	$(1,2,ps)$	$(1,4,wxyz)$	

Figure 8
Method A1 storage of example dictionary

3.1. Method A1

Method A1 uses a total of $nC + (n-1)A$ space in addition to the space required for the n dictionary entries (the space for a dictionary entry is the space required to store the characters that make up the entry). The code representation and the dictionary are stored as a single structure. The prefix code is represented by the corresponding binary tree stored in preorder form. Preorder storage is defined recursively: the root node is stored first, followed by its left subtree stored in preorder form, and then its right subtree in preorder form. In our storage scheme, a leaf node contains a flag bit (set to one, distinguishing between internal nodes and leaves), the length of the corresponding dictionary entry, and the entry itself. For each internal node we store two items, a flag bit (set to zero) and an address. The address component of an internal node is the address of its right subtree. The left subtree for an internal node is stored immediately following the node itself. A tree with n leaves contains $n-1$ internal nodes. Thus, the total storage in addition to the dictionary entries is nC for the leaf nodes and $(n-1)A$ for the internal nodes, assuming that there is a spare bit in the address and length fields. In our application, for which the typical values given in Table 1 apply, the storage requirement is $3n-2$ bytes. Figure 7 shows a Huffman tree for the example dictionary. The codeword for each dictionary entry appears under the entry. We use the convention that left branches are labeled '0' and right branches '1'. Figure 8 gives the corresponding decode data structure. We represent tree nodes as tuples of the form (0,address) or (1,length,entry). The address values are based on allowing 2 bytes for an address ($A = 2$) and 1 byte for each character and each string length ($C = 1$). We assume that the first bit of an address or length field stores the flag bit.

The storage scheme described above allows for simple decoding. For each codeword we begin at the first position of the decode table and we decode one bit at a time. On a 0

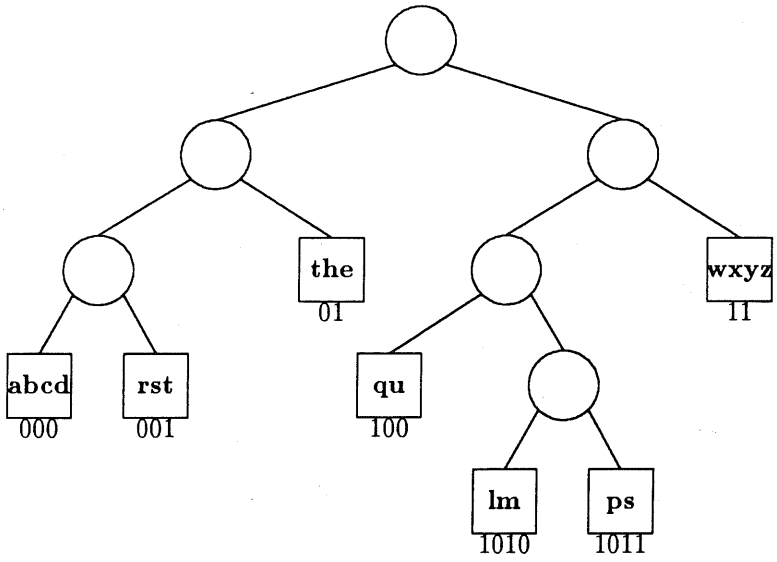


Figure 7
A Huffman tree for example dictionary

dictionary and computing a suboptimal Huffman code. Hankamer's method assumes fixed-length dictionary entries and there is no obvious extension to variable-length entries. This fact coupled with the loss of optimality renders the method inappropriate for our needs. Tanaka gives a finite automaton-based Huffman decoding algorithm [T87]. This method assumes single character dictionary entries. A straightforward modification to allow for variable-length entries is similar to our Method A1 (which follows) in terms of execution speed, but requires approximately 67 percent more memory.

3. Method A

Our first solution to the problem of decoding in restricted memory uses the Huffman code tree to represent the dictionary. However, we do not use the obvious linked implementation in which each internal node contains pointers to its left and right subtrees, as the space requirements of this representation are prohibitive. Instead, Method A1 employs an implicit representation of the tree structure. Method A2 is a variation of Method A1 that provides improved storage utilization.

address	1	3	5	7	12	16	20
contents	$(0,20)$	$(0,16)$	$(0,12)$	$(1,4,abcd)$	$(1,3,rst)$	$(1,3,the)$	$(0,35)$
address	22	24	27	29	32	35	
contents	$(0,27)$	$(1,2,qu)$	$(0,32)$	$(1,2,lm)$	$(1,2,ps)$	$(1,4,wxyz)$	

Figure 8

Method A1 storage of example dictionary

3.1. Method A1

Method A1 uses a total of $nC + (n-1)A$ space in addition to the space required for the n dictionary entries (the space for a dictionary entry is the space required to store the characters that make up the entry). The code representation and the dictionary are stored as a single structure. The prefix code is represented by the corresponding binary tree stored in preorder form. Preorder storage is defined recursively: the root node is stored first, followed by its left subtree stored in preorder form, and then its right subtree in preorder form. In our storage scheme, a leaf node contains a flag bit (set to one, distinguishing between internal nodes and leaves), the length of the corresponding dictionary entry, and the entry itself. For each internal node we store two items, a flag bit (set to zero) and an address. The address component of an internal node is the address of its right subtree. The left subtree for an internal node is stored immediately following the node itself. A tree with n leaves contains $n-1$ internal nodes. Thus, the total storage in addition to the dictionary entries is nC for the leaf nodes and $(n-1)A$ for the internal nodes, assuming that there is a spare bit in the address and length fields. In our application, for which the typical values given in Table 1 apply, the storage requirement is $3n-2$ bytes. Figure 7 shows a Huffman tree for the example dictionary. The codeword for each dictionary entry appears under the entry. We use the convention that left branches are labeled '0' and right branches '1'. Figure 8 gives the corresponding decode data structure. We represent tree nodes as tuples of the form (0,address) or (1,length,entry). The address values are based on allowing 2 bytes for an address ($A = 2$) and 1 byte for each character and each string length ($C = 1$). We assume that the first bit of an address or length field stores the flag bit.

The storage scheme described above allows for simple decoding. For each codeword we begin at the first position of the decode table and we decode one bit at a time. On a 0

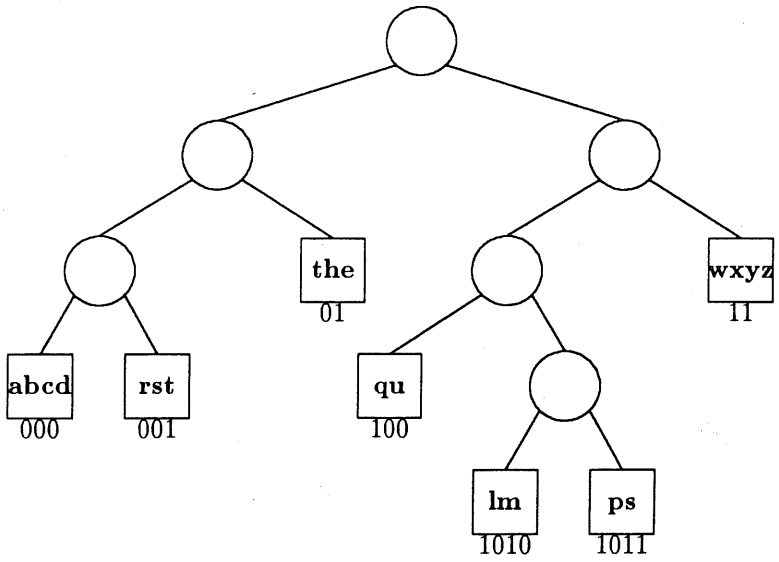


Figure 7
A Huffman tree for example dictionary

dictionary and computing a suboptimal Huffman code. Hankamer's method assumes fixed-length dictionary entries and there is no obvious extension to variable-length entries. This fact coupled with the loss of optimality renders the method inappropriate for our needs. Tanaka gives a finite automaton-based Huffman decoding algorithm [T87]. This method assumes single character dictionary entries. A straightforward modification to allow for variable-length entries is similar to our Method A1 (which follows) in terms of execution speed, but requires approximately 67 percent more memory.

3. Method A

Our first solution to the problem of decoding in restricted memory uses the Huffman code tree to represent the dictionary. However, we do not use the obvious linked implementation in which each internal node contains pointers to its left and right subtrees, as the space requirements of this representation are prohibitive. Instead, Method A1 employs an implicit representation of the tree structure. Method A2 is a variation of Method A1 that provides improved storage utilization.

address	1	3	5	7	12	16	20
contents	$(0,20)$	$(0,16)$	$(0,12)$	$(1,4,abcd)$	$(1,3,rst)$	$(1,3,the)$	$(0,35)$
address	22	24	27	29	32	35	
contents	$(0,27)$	$(1,2,qu)$	$(0,32)$	$(1,2,lm)$	$(1,2,ps)$	$(1,4,wxyz)$	

Figure 8
Method A1 storage of example dictionary

3.1. Method A1

Method A1 uses a total of $nC + (n-1)A$ space in addition to the space required for the n dictionary entries (the space for a dictionary entry is the space required to store the characters that make up the entry). The code representation and the dictionary are stored as a single structure. The prefix code is represented by the corresponding binary tree stored in preorder form. Preorder storage is defined recursively: the root node is stored first, followed by its left subtree stored in preorder form, and then its right subtree in preorder form. In our storage scheme, a leaf node contains a flag bit (set to one, distinguishing between internal nodes and leaves), the length of the corresponding dictionary entry, and the entry itself. For each internal node we store two items, a flag bit (set to zero) and an address. The address component of an internal node is the address of its right subtree. The left subtree for an internal node is stored immediately following the node itself. A tree with n leaves contains $n-1$ internal nodes. Thus, the total storage in addition to the dictionary entries is nC for the leaf nodes and $(n-1)A$ for the internal nodes, assuming that there is a spare bit in the address and length fields. In our application, for which the typical values given in Table 1 apply, the storage requirement is $3n-2$ bytes. Figure 7 shows a Huffman tree for the example dictionary. The codeword for each dictionary entry appears under the entry. We use the convention that left branches are labeled '0' and right branches '1'. Figure 8 gives the corresponding decode data structure. We represent tree nodes as tuples of the form (0,address) or (1,length,entry). The address values are based on allowing 2 bytes for an address ($A = 2$) and 1 byte for each character and each string length ($C = 1$). We assume that the first bit of an address or length field stores the flag bit.

The storage scheme described above allows for simple decoding. For each codeword we begin at the first position of the decode table and we decode one bit at a time. On a 0

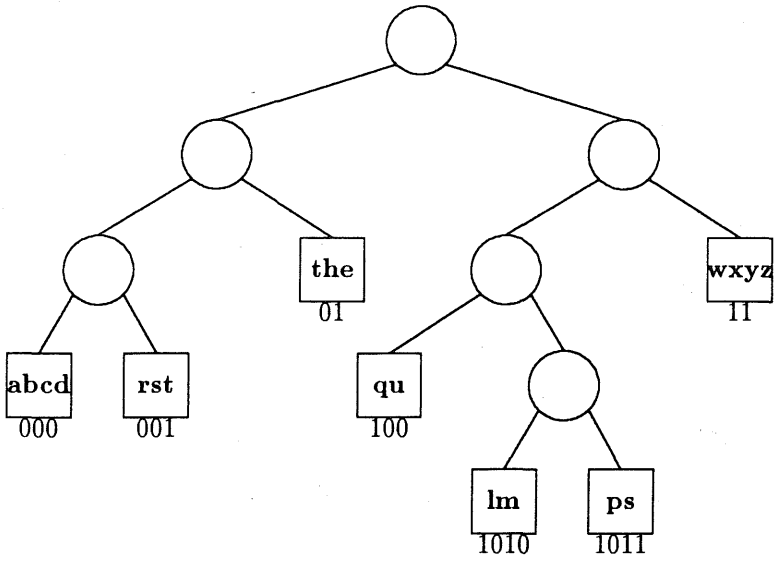


Figure 7
A Huffman tree for example dictionary

dictionary and computing a suboptimal Huffman code. Hankamer's method assumes fixed-length dictionary entries and there is no obvious extension to variable-length entries. This fact coupled with the loss of optimality renders the method inappropriate for our needs. Tanaka gives a finite automaton-based Huffman decoding algorithm [T87]. This method assumes single character dictionary entries. A straightforward modification to allow for variable-length entries is similar to our Method A1 (which follows) in terms of execution speed, but requires approximately 67 percent more memory.

3. Method A

Our first solution to the problem of decoding in restricted memory uses the Huffman code tree to represent the dictionary. However, we do not use the obvious linked implementation in which each internal node contains pointers to its left and right subtrees, as the space requirements of this representation are prohibitive. Instead, Method A1 employs an implicit representation of the tree structure. Method A2 is a variation of Method A1 that provides improved storage utilization.

address	1	3	5	7	12	16	20
contents	$(0,20)$	$(0,16)$	$(0,12)$	$(1,4,abcd)$	$(1,3,rst)$	$(1,3,the)$	$(0,35)$
address	22	24	27	29	32	35	
contents	$(0,27)$	$(1,2,qu)$	$(0,32)$	$(1,2,lm)$	$(1,2,ps)$	$(1,4,wxyz)$	

Figure 8
Method A1 storage of example dictionary

3.1. Method A1

Method A1 uses a total of $nC + (n-1)A$ space in addition to the space required for the n dictionary entries (the space for a dictionary entry is the space required to store the characters that make up the entry). The code representation and the dictionary are stored as a single structure. The prefix code is represented by the corresponding binary tree stored in preorder form. Preorder storage is defined recursively: the root node is stored first, followed by its left subtree stored in preorder form, and then its right subtree in preorder form. In our storage scheme, a leaf node contains a flag bit (set to one, distinguishing between internal nodes and leaves), the length of the corresponding dictionary entry, and the entry itself. For each internal node we store two items, a flag bit (set to zero) and an address. The address component of an internal node is the address of its right subtree. The left subtree for an internal node is stored immediately following the node itself. A tree with n leaves contains $n-1$ internal nodes. Thus, the total storage in addition to the dictionary entries is nC for the leaf nodes and $(n-1)A$ for the internal nodes, assuming that there is a spare bit in the address and length fields. In our application, for which the typical values given in Table 1 apply, the storage requirement is $3n-2$ bytes. Figure 7 shows a Huffman tree for the example dictionary. The codeword for each dictionary entry appears under the entry. We use the convention that left branches are labeled '0' and right branches '1'. Figure 8 gives the corresponding decode data structure. We represent tree nodes as tuples of the form (0,address) or (1,length,entry). The address values are based on allowing 2 bytes for an address ($A = 2$) and 1 byte for each character and each string length ($C = 1$). We assume that the first bit of an address or length field stores the flag bit.

The storage scheme described above allows for simple decoding. For each codeword we begin at the first position of the decode table and we decode one bit at a time. On a 0

```
k ← 1
repeat
        receive bit
        if bit = 0
        then k ← k + A
        else k ← address(k)
        flag_value ← flag(k)
until flag_value = 1
append contents of memory locations k ... k + length(k) − 1 to the decoded output
```

Figure 9
Method A1 decoding

bit we move from an internal node to its left child by advancing over the address field. On a 1 bit we use the address field to move to the right subtree of the current internal node. We continue to decode bits until a flag value of 1 is encountered, indicating a leaf node. At this point we have detected the end of a codeword and located the corresponding dictionary entry. The dictionary entry is appended to the decoded output, and we return to the first position of the decode table ready to decode the next codeword. The operations given in Figure 9 are performed for each codeword in the encoded source. We use $address(k)$ to represent the address component of an internal node k, $flag(k)$ to represent the flag component of any node k, and $length(k)$ to represent the length component of a leaf node k.

The encoder transmits the tree to the decoder in the form we have described. Thus, the representation overhead associated with Method A1 is $nC + (n-1)A$, and the delay (time for setup) consists of the time necessary to receive and store the tree.

3.2. Method A2

The storage requirement of Method A1 can be improved in some cases by exploiting the fact that the length values need not be stored in the decode data structure. The key observation that allows us to eliminate the string lengths is that we can find the length of an entry by subtracting its starting address from the starting address of its preorder successor. The starting address of any leaf node's preorder successor can be found easily, trivially, in fact, if the leaf, x, is a left child of its parent. In this case, the preorder successor of x is its sibling, and the address of the sibling is stored in x's parent node. In the other case, when x is a right child, we can walk from x to its preorder successor as

follows: we walk up '1' branches until we reach a node that is not a right child; at this point, we walk up a single '0' branch and then down a '1' (right) branch. In other words, the preorder successor of x is the right child of the lowest internal node from which we follow a '0' (left) branch to x. This characterization is also valid when x is a left child, since x's parent is the lowest internal node from which we follow a left branch to x, and x's preorder successor is the right child of this (parent) node. The only node for which the above characterization is not valid is the final node in the preorder listing. This node lies on a path from the root consisting of only right branches, and it has no preorder successor. So that we can decode this final node, we store the address of its (nonexistent) preorder successor in address 0 of the decode data structure, ahead of the preorder representation of the decode tree. Thus we store n addresses in Method A2 instead of the $n-1$ addresses used in Method A1.

In the Method A1 data structure, address values are coupled with flag bits to represent internal nodes, and length values are coupled with flag bits and combined with character strings to represent leaf nodes. The coupling is accomplished by using the leading bit of the address or length value for storing the flag. In eliminating the length value from a leaf node, we are presented with the problem of how to store the flag bit. The best solution to this problem is to couple the flag bit with the leading character of the dictionary entry. In order for this to be possible, we must be able to store characters in $b-1$ bits (where b is the number of bits per byte). This assumption may be reasonable on machines with 8-bit bytes where the application involves storing or transmitting text. The printable characters typical of text files can be represented in seven bits. Under this assumption, the storage requirement of Method A2 becomes nA, as compared with $(n-1)A+nC$ for Method A1. Using the typical values given in Table 1, we have $2n$ bytes for Method A2, as compared with $3n-2$ bytes for Method A1.

If the assumption of a spare bit in character storage is not valid, eliminating the lengths may not provide an improvement in storage utilization. Since high-level languages have the byte as the atomic unit of addressable memory, we are forced to store the flag in a byte when neither the length field nor the character field can accommodate it. If string lengths can be stored in a single byte ($C=1$) with a spare bit, we gain nothing by storing a one-byte flag instead of a one-byte (flag,length) pair. In fact, the storage requirement for Method A2 would be $nA+n$ bytes as compared with $(n-1)A+n$ bytes for Method A1. In a case where lengths require more than one byte of storage ($C \geq 2$), however, the one-byte flag would be an improvement over the C-byte (flag,length) pair. In this case, Method A1

address	0	2	4	6	8	12	15	18
contents	34	$(0, 18)$	$(0, 15)$	$(0, 12)$	$(1, abcd)$	$(1, rst)$	$(1, the)$	$(0, 30)$
address	20	22	24	26	28	30		
contents	$(0, 24)$	$(1, qu)$	$(0, 28)$	$(1, lm)$	$(1, ps)$	$(1, wxyz)$		

Figure 10
Method A2 storage of Figure 6 example

requires $(n-1)A+nC$ bytes of storage and Method A2 requires only $nA+n$. In addition, the use of the (flag,length) coupling depends on the assumption that lengths can be stored in such a way as to provide a spare bit for the flag. If this assumption is not valid, storing the flag alone will provide a space improvement over storing the (flag,length) pair in $C+1$ bytes. In summary, the elimination of the length values from the Method A1 data structure is not guaranteed to provide improved storage utilization, but does so under fairly general conditions. In fact, Method A2 will be superior to Method A1 unless characters require all b bits in a byte and string lengths require at most $b-1$ bits. And in this case Method A2 will be inferior by most $A+1$ bytes!

In Figure 10 we give the Method A2 data structure for the example dictionary of Figure 6 under the assumption that each character contains a spare bit that can be used for the flag value. We assume that address fields also contain the spare bit and that $A = 2$. We represent internal nodes as (flag,address) pairs and leaf nodes as (flag,entry) pairs.

Using the Method A2 data structure to decode is very similar to using the Method A1 structure. The only difference is that, in addition to the address of the dictionary entry being decoded, we are also looking for the address of its preorder successor. The instructions in Figure 11 are performed for each codeword. We use $address(k)$ and $flag(k)$ as in Method A1 and p represents the current candidate for the address of the preorder successor. We use the notation $contents(0)$ to retrieve the successor of the last node in the preorder listing from memory location 0. Decode speed is very similar to that of Method A1. The only extra time is due to storing an address in p for each 0 bit.

The encoder transmits the tree to the decoder in the form we have described. Thus, assuming a spare bit in character bytes, the representation overhead for Method A2 is

```
p ← contents(0)
k ← 2
repeat
        receive bit
        if bit = 0
        then p ← address(k)
                k ← k + A
        else k ← address(k)
        flag_value ← flag(k)
until flag_value = 1
append contents of memory locations k ... p − 1 to the decoded output
```

Figure 11
Method A2 decoding

nA and the set-up time consists of the time necessary to receive and store the tree. Both representation overhead and set-up time are smaller for Method A2 than for Method A1. Tables 2 and 3 present space and time comparisons of our methods. The data for Method A1 presumes the spare bit in the address and length bytes, and for Method A2 the spare bit in character bytes is assumed.

4. Method B

The second method we discuss is based on the concept of a canonical Huffman code defined by Schwartz and Kallick [SK64] and by Connell [C73]. We describe this concept first and then our implementation of it. The essence of the canonical code concept is that Huffman's algorithm is needed only to compute the lengths of the codewords to be mapped to the dictionary entries. Once lengths are determined, actual codewords may be specified in many ways. The only necessary condition is that they satisfy the prefix property. This is true for prefix codes in general.

In addition to the fact that there are many ways of forming codewords of appropriate lengths, there are cases in which the Huffman algorithm does not uniquely determine these lengths due to the arbitrary choice among equal minimum weights. As an example, codes with codeword lengths of $\{1, 2, 3, 4, 4\}$ and of $\{2, 2, 2, 3, 3\}$ both yield the same

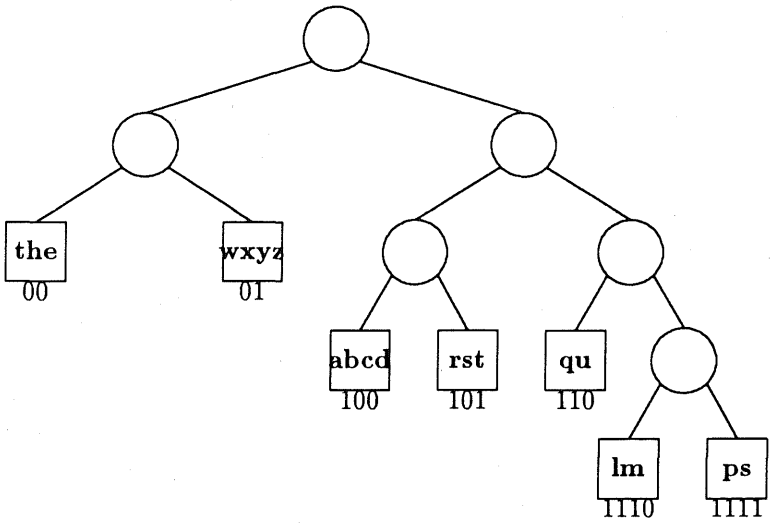


Figure 12
The canonical Huffman code tree for Figure 6 example

average codeword length for a source with probabilities $\{.4, .2, .2, .1, .1\}$. Schwartz defines a variation of the Huffman algorithm that performs *bottom merging*, that is, orders a new parent node above existing nodes of the same weight and always merges the last two weights in the list [S64]. The code constructed is the Huffman code with minimum values of maximum codeword length ($\max\{l_i\}$) and total codeword length ($\sum l_i$). Schwartz and Kallick's canonical Huffman code is constructed using bottom merging [SK64].

Intuitively, the canonical code may be viewed as one that builds the prefix code tree from left to right in increasing order of depth (i.e., codeword length) with the convention that each leaf is placed at the "first" position (from left to right) available to it. The example dictionary has codeword length sequence [2,2,3,3,3,4,4]. In constructing the canonical code, the first codeword of length two is placed at the left edge of level two of the tree. Using the convention that left branches are labeled with 0 and right branches with 1, the first codeword is 00. The second codeword of length two is the sibling of the first, 01. The first codeword of length three is placed at the first available position on level three of the tree. Level three is filled from left to right by placing codewords 100, 101, and 110. The length-four codewords, 1110 and 1111, complete the tree. The canonical code tree for the example dictionary is given in Figure 12. The codeword for each dictionary entry appears under the entry.

The canonical code possesses some nice mathematical properties. The codewords of a given length are consecutive binary numbers. The first codeword of length l, c_l, is related to the last codeword of length $l-1$, d_{l-1}, by the equation $c_l = 2(d_{l-1}+1)$. In other words, the first codeword of length l is obtained from the last codeword of length $l-1$ by adding 1 to the binary number represented by d_{l-1} and shifting that binary number left once. In the case where some lengths are unused, as in [1,3,3,3,4,4], the codewords of length 3 are consecutive binary numbers as are the codewords of length 4. The function that computes the first length-3 codeword from the length-1 codeword is $2(2(d_1+1))$. That is, to move down two levels in the tree from level 1 to level 3, two shifts are required. For the length sequence [1,3,3,3,4,4], the canonical code is $\{0, 100, 101, 110, 1110, 1111\}$. Every canonical code has a string of zeros as its first (shortest) codeword and a string of ones as its last (longest) codeword. We say that a canonical code has the *numerical sequence property.*

We now discuss the way in which the numerical sequence property contributes to reducing memory requirements. First, the canonical code eliminates the need for the encoder to transmit to the decoder an explicit representation of the tree, since the length sequence is sufficient to define the tree. We represent the length sequence as a list consisting of:

(1) *min*, the length of the shortest codeword,

(2) *max*, the length of the longest codeword, and

(3) the number of codewords of each length.

The first example above is, thus, represented by 2,4,2,3,2 and the second by 1,4,1,0,3,2. In most cases this representation is more compact than a list of the lengths of all of the codewords. If the encoder uses the length list to define the code, the size of the representation is $2B + LM$ where $L = max - min + 1$, M represents the number of bytes required to store the maximum number of codewords of any given length, and B the number of bytes required to store the length of a codeword. We will show that the data structure needed by the decoder can be constructed efficiently given the length list.

In addition to providing a compact representation of the code, the numerical sequence property may be used to index into the data dictionary. This is done through the use of two small tables, *limit* and *base*. Each of these tables is indexed from *min* to *max*. The *limit* table is used in decoding to detect the end of a codeword. The entry *limit*[i] contains

the value of the largest codeword of length at most i. The numerical sequence property guarantees that the numerical value of a codeword of length i is greater than the value of any shorter codeword. Thus if the binary value of a string of i bits is greater than $limit[i]$ the string is not a codeword but a prefix of a codeword. The decoder reads min bits from the coded text. If the binary value of this bit string is less than or equal to $limit[min]$ the bit string represents a codeword. If the value of the first min bits is greater than $limit[min]$ the decoder reads another bit, updates the value of the bit string, and compares that value to $limit[min+1]$. This process continues until the value of the bit string of length i is less than or equal to $limit[i]$ for some i. At this point we have recognized a codeword. Once the end of a codeword is detected the *base* table may be used to locate the corresponding dictionary entry. The *base* table as defined by Connell maps a codeword value onto the relative position of the corresponding dictionary entry in a list of dictionary entries [C73].

The information provided by the *limit* and *base* tables is sufficient to allow decoding if the entries of the data dictionary are all of the same length. For variable-length entries, however, we need the address of the appropriate entry, not an index. We present two solutions to this problem. We comment that tables *limit* and *base* as defined by Connell are redundant with respect to one another [C73]. That is, the information contained in the *base* table can be extracted from the *limit* table entries. The *base* table, however, can be represented in very little space and contributes substantially to the clarity of exposition of our methods. Eliminating the *base* table also results in slower decoding, therefore we maintain the *base* table.

4.1. Method B1

Method B1 adapts Connell's *base* table method to allow for variable-length dictionary entries by introducing an *address* table indexed from 1 to $n+1$. The value of $address[k]$ is the address of the first character of the k^{th} dictionary entry. The entries are stored in a *string* table that is organized in the following way: entries are stored in nondecreasing order by codeword length and the block of entries with codeword length i is stored in order of decreasing codeword value. In terms of the prefix tree we store the dictionary in *modified level order*, that is, in increasing order by level and in order from right to left on each level (of course we are storing only the leaves of the prefix tree). The *base* table provides pointers into the *address* table. That is, $base[i]$ contains x such that $address[x]$ is the starting address of the block of dictionary entries with codeword length i. When a

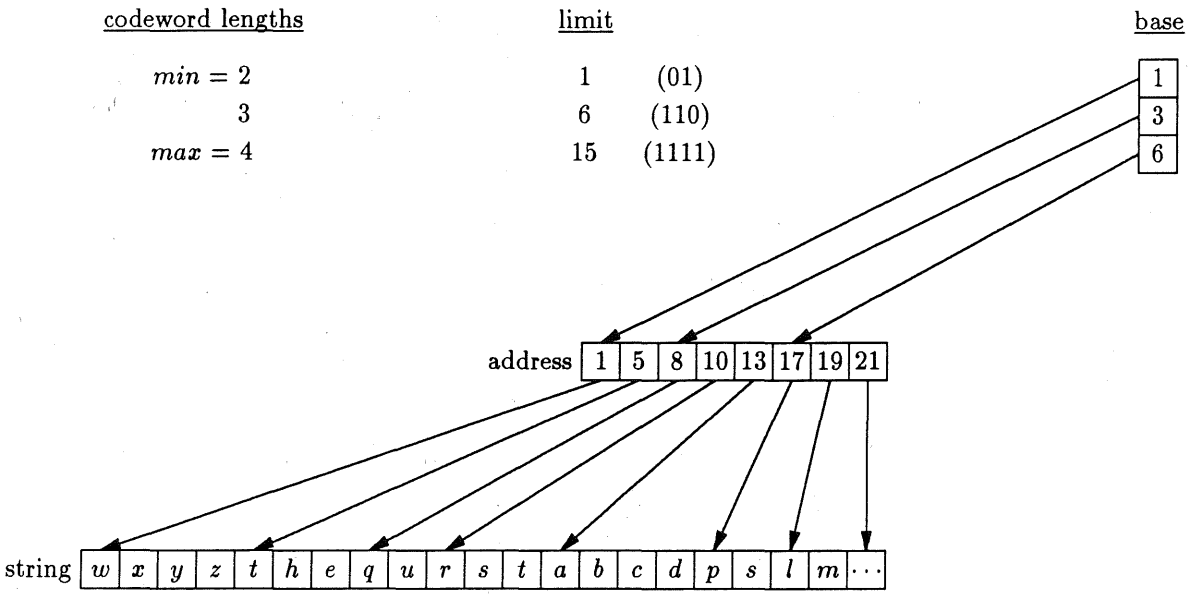


Figure 13

Method B1 data structure for Figure 6 example

codeword c of length i is recognized, $limit[i] - value(c)$* provides an offset in the list of codewords of length i. Thus $p = base[i] + limit[i] - value(c)$ is the subscript in the *address* table at which the beginning of the corresponding dictionary entry is stored. The length of the entry is given by $address[p+1] - address[p]$. The address and length of the entry are all we need to append the entry to the output of the decoder. The storage requirement at decode time consists of LV for the *limit* table (*limit* contains codeword values), LN for the *base* table (*base* contains subscripts from 1 to $n+1$), and $(n+1)A$ for the *address* table. In most cases we expect $LV + LN + (n+1)A$ to be an improvement over the $nC + (n-1)A$ requirement of Method A1. In practice L is generally $O(\lg n)$ while a typical value of L is 13. Therefore Method A1 requires $3n - 2$ bytes and Method B1 $2n + 54$ in a typical application. The storage requirement of Method B1 will always be greater than the $2n$ requirement of Method A2, so that Method B1 provides no improvement in an application in which character bytes contain an unused bit. In terms of translation time, Method B1 is expected to be a little bit slower than the A Methods, but not significantly slower.

* $value(c)$ is the binary value of codeword c

Figure 13 gives the Method B1 data structure for the example dictionary. The addresses represent byte addresses of dictionary entries. We assume that the starting address is 1, and that each character of an entry occupies 1 byte. Tables 2 and 3 provide space and time comparisons of Methods A1, A2, and B1.

The encoder transmits the length list, the strings, and their lengths as a preface to the encoded text. Thus, the representation overhead is $2B + LM + nC$. The representation is transmitted in the following form: first, *min* and *max*; then for each codeword length i (from *min* to *max*), n_i followed by n_i $(length, str)$ pairs. Each n_i represents the number of dictionary entries with codeword length i and each $(length, str)$ pair gives the number of characters in a dictionary entry followed by the character string itself. The entries with codeword length i are listed in order of decreasing codeword value. The decoder performs the calculations given in Figure 14 to set up the decode data structure. In addition to the time required to receive the data, the decoder performs $\theta(n)$ operations in setting up the *address* table and $\theta(L)$ operations in constructing tables *limit* and *base*.

4.2. Method B2

We now present a modification of Method B1 that can provide space utilization superior to that of Method A2. Method B2 is actually a collection of methods, parameterized by a variable k. The time-space compromise that best fits the requirements of a particular application can be selected by fixing an appropriate value of k. The improvement in Method B2 over Method B1 is achieved by storing fewer than n address values. The value of the parameter k determines what fraction of the n address values are stored. Method B2 uses the *limit* and *base* tables exactly as in Method B1. The dictionary is represented by three tables. The first table, *string*, contains the dictionary entries stored as in Method B1 (i.e., in modified level order). The second table, *address*, is indexed from 1 to $\lfloor \frac{n}{k} \rfloor$ and stores the address of every k^{th} dictionary entry, with $address[j]$ containing the address of entry jk. The third table, *len*, is indexed from 1 to $n - \lfloor \frac{n}{k} \rfloor$ and contains string lengths. Thus the space requirements of Method B2 are: $LV + LN$ for the *limit* and *base* tables, $\lfloor \frac{n}{k} \rfloor A$ for the *address* table, and $(n - \lfloor \frac{n}{k} \rfloor)C$ for the *len* table.

The *limit* table is used to recognize codewords as in Method B1. The *base* table again yields an index into the list of dictionary entries. If $base[i] = x$ then the x^{th} dictionary entry is the first entry (in modified level order) with codeword of length i. When a codeword c of length i has been decoded, we use $p = base[i] + limit[i] - value(c) - 1$ to find the corresponding dictionary entry. If $p \bmod k = 0$, the address of the first character

```
s ← 1
a ← 1
receive min, max
for i ← min to max do
        receive n_i
        if i = min
        then base[min] ← 1
             limit[min] ← n_min − 1
        else base[i] ← base[i − 1] + n_{i−1}
             limit[i] ← 2(limit[i − 1] + 1) + n_i − 1
        for j ← 1 to n_i do
                receive length, str
                store str in string[s ··· s + length − 1]
                address[a] ← s
                a ← a + 1
                s ← s + length
        endfor
endfor
address[a] ← s
```

Figure 14
Method B1 set up

of the entry is stored in $address[\frac{p}{k}]$. If $p \bmod k \neq (k-1)$, the length of entry p is stored in $len[p - \lfloor \frac{p}{k} \rfloor + 1]$. Thus, when $p \bmod k = 0$, both the address and the length of the corresponding dictionary entry are stored in the decode data structure. When $p \bmod k \neq 0$, $address[\lfloor \frac{p}{k} \rfloor]$ is a pointer to the block of k entries that includes the one we seek. We "walk" along this block until we find the entry corresponding to c. This walk can be viewed as a sequence of "jumps" that use the *len* values to jump over entries. The number of jumps is given by $p \bmod k$ so that the maximum number of jumps is $k-1$. If $p \bmod k \neq (k-1)$, the length of the entry is stored in the *len* table. Otherwise, the length of the entry is computed from the starting address of its successor in the modified level order listing (i.e.,

$p \leftarrow base[i] + limit[i] - value(c) - 1$

$q \leftarrow \lfloor \frac{p}{k} \rfloor$

if $q = 0$

then $start \leftarrow 1$

else $start \leftarrow address[q]$

$r \leftarrow p \bmod k$

$t \leftarrow p - q$

for $i \leftarrow 1$ to r do

$start \leftarrow start + len[t - i + 1]$

endfor

if $r \neq k - 1$

then $length \leftarrow len[t + 1]$

else $length \leftarrow address[q + 1] - start$

Figure 15

Computing *start* and *length*

$address[\lfloor \frac{p}{k} \rfloor + 1]$). The calculations given in Figure 15 provide the starting address *start* and the *length* corresponding to any index p.

As in Method B1, the encoder transmits the length list, the strings, and their lengths. Thus the representation overhead is $2B + LM + nC$. Tables *limit* and *base* are built exactly as in Method B1. The code in Figure 16 sets up the data structures *limit*, *base*, *len* and *address*. The set-up time is again $\theta(n) + \theta(L)$.

Figure 17 gives the Method B2 data structure for the example of Figure 6 with $k = 2$. A comparison with the other methods is provided in Tables 2 and 3. We note that if $k = 1$ the storage requirement for Method B2 reduces to the requirement for Method B1.

We provide a second example for Method B2 in Figure 18. The data structure for an example with a larger dictionary and $k = 3$ is given. The reader can use the *limit* table values to verify that the codewords for {*wxyz*, *the*, *qu*, *rst*, *abcd*, *ps*, *lm*, *out*, *rt*} are {0, 110, 101, 100, 11110, 11101, 11100, 111111, 111110}.

```
s ← 1
a ← 1
l ← 1
count ← 0
receive min, max
for i ← min to max do
        receive n_i
        if i = min
        then base[min] ← 1
             limit[min] ← n_min − 1
        else base[i] ← base[i − 1] + n_{i−1}
             limit[i] ← 2(limit[i − 1] + 1) + n_i − 1
        for j ← 1 to n_i do
             receive length, str
             store str in string[s ··· s + length − 1]
             if count mod k ≠ k − 1
             then len[l] ← length
                  l ← l + 1
                  if count ≠ 0 and count mod k = 0
                  then address[a] ← s
                       a ← a + 1
             count ← count + 1
             s ← s + length
        endfor
endfor
if count mod k = 0
then address[a] ← s
```

Figure 16
Method B2 set up

codeword lengths	limit		base	len
$min = 2$	1	(01)	1	4 ($wxyz$)
3	6	(110)	3	2 (qu)
$max = 4$	15	(1111)	6	4 ($abcd$)
				2 (lm)

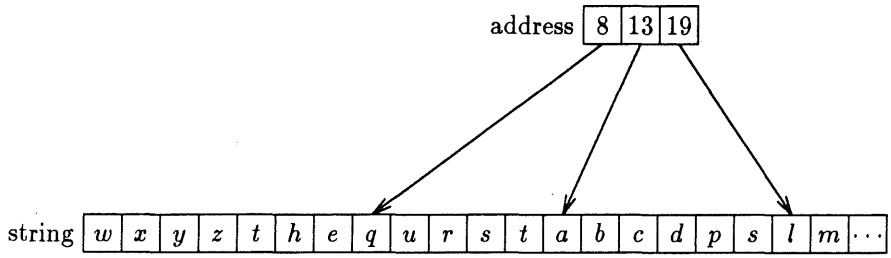


Figure 17
Method B2 data structure for Figure 6 example ($k = 2$)

The parameter k determines the decode speed of Method B2 as well as its storage requirement. The maximum number of jumps determines the worst case time for appending one dictionary entry to the output. The maximum number of jumps is $k-1$. It is important to recognize that the time-space tradeoff provided by Method B2 is nonlinear. When $k = 1$, Method B2 stores n addresses. When $k = n$, Method B2 stores 1 address and $n-1$ lengths. Assuming $A = 2$ and $C = 1$, the choice $k = 1$ requires $2n$ bytes of storage, and the choice $k = n$ requires $n + 1$ bytes. When $k = 2$, the storage requirement is $1.5n$ bytes, essentially midway between the requirement for $k = 1$ and that for $k = n$. However, the choice of $k = 2$ may result in decode speed much closer to that provided by $k = 1$ than that provided by $k = n$. The extra decode time required by Method B2 (as compared to Method B1) is proportional to the number of jumps. When $k = n$, only one address is stored. Thus, the first codeword (in modified level order) can be decoded with no jumps, the second requires 1 jump, and in general the j^{th} requires $j - 1$ jumps. The maximum number of jumps required to decode a single codeword is $n - 1$. Employing Method B2 with $k = 2$ reduces the maximum number of jumps to just one. If we compare the use of $k = n$ with the use of $k = 1$, we see that by doubling the space requirement we eliminate the need to

codeword lengths	limit		base	len
$min = 1$	0	(0)	1	4 (*wxyz*)
2	1			3 (*the*)
3	6	(110)	2	3 (*rst*)
4	13			4 (*abcd*)
5	30	(11110)	5	2 (*lm*)
$max = 6$	63	(111111)	8	3 (*out*)

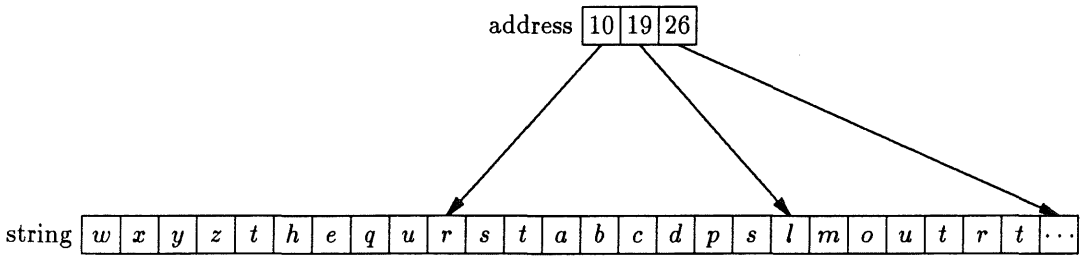


Figure 18
Method B2 data structure for example with $k = 3$

jump since every address is stored. We can reduce the maximum number of jumps to one, however, at a cost of only 50 percent extra space. In general, a space increase of $\frac{1}{k}$ of the $k = n$ requirement (which stores only a single address and all n string lengths) imposes a ceiling of $k-1$ on the number of jumps. In practice a k value of about 4 or 5 is reasonable.

Before we present a summary of the performance of our methods, we return to the observation made earlier that, for Method B1 the *limit* and *base* tables are mutually redundant. The index into the list of dictionary entries provided by the *base* table can be computed at decode time from the *limit* values. We note that this is not true for Method B2 in which *base* table values are addresses rather than list positions. To compute the *base* values from the *limit* values, we use the fact that $base[i] = 1 + \sum_{j=min}^{i-1} n_j$ where n_j is the number of codewords of length j. The value of n_j can be computed from the largest codeword of length j ($limit[j]$) and the smallest codeword of length

```
receive bit
codeword_val ← bit
for i = 2 to min do
        receive bit
        codeword_val ← 2 * codeword_val + bit
endfor
len ← min
base ← 1
while codeword_val > limit[len] do
        receive bit
        codeword_val ← 2 * codeword_val + bit
        if len = min
        then base ← base + limit[len] + 1
        else base ← base + limit[len] − 2 * limit[len − 1] − 1
        len ← len + 1
endwhile
p ← base + limit[len] − codeword_val
```

Figure 19
Computing p without use of *base* table

j $(2 * (limit[j-1]+1))$. Combining these facts and simplifying, we have $base[i] = limit[i-1] - \sum_{j=min}^{i-2} limit[j] + min - i + 3$. Computing *base* values at decode time requires a few more arithmetic operations per bit. On the other hand, decode storage requirements are reduced by the LN bytes of memory needed to store the *base* table. The change in the use of the *base* table has no effect on representation overhead while a very small amount of set-up time is saved by not computing the *base* table. We note that, in the typical case, eliminating the *base* table saves 32 bytes of memory at the expense of several extra arithmetic operations per bit decoded. Unless available memory is severely constrained or the values of L and N are atypical, the time-space tradeoff afforded by this modification is not advantageous. The code given in Figure 19 demonstrates that the decoder can find the position p used in Method B1 to index into the *address* table.

Method	Representation Overhead	Decode Space Requirements	Decode Space in "Typical" Application
A1	$(n-1)A+nC$	$(n-1)A+nC$	$3n-2$
A2	nA	nA	$2n$
B1	$2B+LM+nC$	$(n+1)A+LV+LN$	$2n+54$
B2	$2B+LM+nC$	$\lfloor \frac{n}{k} \rfloor A+(n-\lfloor \frac{n}{k} \rfloor)C+LV+LN$	$1.2n+52$

Table 2
Space comparison of methods

Method	Receiving Time for Code Description	Additional Set-up Time	Relative Decode Time
A1	$(n-1)A+nC$	*none*	*very fast*
A2	nA	*none*	*very fast*
B1	$2B+LM+nC$	c_1L+c_2n	*very fast*
B2	$2B+LM+nC$	c_1L+c_2n	*fast*

Table 3
Time comparison of methods

We present a summary of the performance of our methods in Tables 2 and 3. The space requirement we give for Method B1 includes the LN bytes for the base table. The typical values are those given in Table 1 with the addition of $k = 5$. In the second column of Table 3, labeled "Receiving Time for Code Description", we give the number of bytes transmitted for the code description. Clearly the time required to receive the data is proportional to its size. In column three of Table 3, c_1 and c_2 represent small constants. We note that, while the A Methods require no additional set-up time, their code descriptions are almost guaranteed to be longer than those of the B Methods, so that the larger receiving time requirement offsets the savings in set-up time.

5. Additional Implementation Considerations

5.1. Reducing transmission time

We consider several issues associated with the representation of:

(1a) the stream of characters,

(1b) information needed to reconstruct the dictionary from the character stream, and

(2) the prefix code.

Our discussions have focused on the way the representation is stored in the decoder and the way it is used to decode the message. We now make some observations on the way in which it is transmitted.

We have assumed that the characters of the dictionary are stored one character per byte in our decode data structures. It is not necessary to respect byte boundaries in transmitting the stream of characters. The stream of characters would typically be represented in 7- or 8-bit ASCII. If the dictionary is very large, however, it may be significantly more efficient to employ a variable-length coding technique. The canonical Huffman code can be used at very low cost for encoding single characters since only tables *limit* and *base* and an array of characters in modified level order are required for decoding.

In Tables 2 and 3 we include nC bytes in the representation overhead for the lengths of the dictionary entries. We observe, first, that it is not necessary that an integer number of bytes be used to transmit a string length. In addition, if the lengths of the entries vary across a wide range, we can do much better than nC bytes by using a variable-length representation of the integers such as the Fibonacci codes described in Section 1.2 of Chapter 2. If the lengths of dictionary entries vary from l_1 to l_2, a fixed-length representation requires $\lg l_2$ bits for each length. The variable-length codes represent small lengths in fewer than $\lg l_2$ bits, but large length values require more bits. The variable-length code is justified, then, if dictionary entries are short on average. For Methods B1 and B2, in which the prefix code is represented by a length list, the same variable-length coding can be applied to codeword lengths. Codeword lengths are expected to be short. It is likely that most of them can be represented in less than one byte. The Fibonacci codes are simple to encode and decode in place, and are well suited for representing integers.

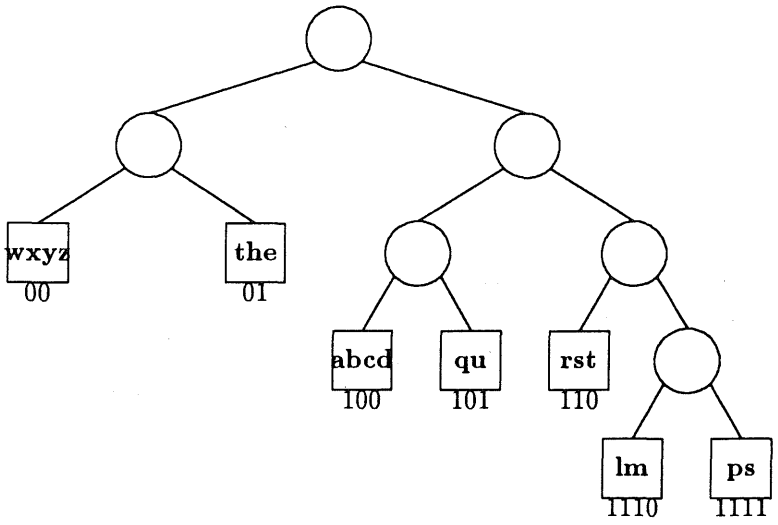


Figure 20
The B2-optimal tree for Figure 6 example

5.2. Reducing decode time

Another implementation detail worthy of mention is one that can reduce decode time for Method B2. Just as the canonical Huffman code can be viewed as a refinement of standard Huffman coding (in that it selects a particular code tree among multiple optimal trees), we present a further refinement of the canonical code, which we call the *B2-optimal canonical code.* We note, first, that while the canonical code specifies a code tree, it leaves open the question of how to assign the n_i codewords of length i to the n_i dictionary entries. We specify this assignment so as to minimize the average number of jumps (thus a B2-optimal canonical code is one that minimizes decode time).

The observation which allows us to minimize the average number of jumps is that, on each level of the code tree, the number of jumps required to decode a dictionary entry increases as its position in the modified level order increases and as its position increases from right to left in modified level order. Thus we assign the dictionary entries to the canonical code tree so that the frequencies of dictionary entries decrease from right to left. This way, the most frequent entry on each level will require the smallest number of jumps to decode. Figure 20 shows the B2-optimal canonical Huffman code tree for the example dictionary of Figure 6. The B2-optimal code depends on the parameter k

and on the interplay between k and the number of codewords of each length. Figure 17 shows that decoding any of **wxyz**, **qu**, **abcd**, or **lm** requires no jumps and that decoding either **the**, **rst**, or **ps** requires one jump. The B2-optimal code reverses the positions of **wxyz** and **the** so that the entry with higher frequency can be decoded without jumps. Two of the level-three entries can be decoded without jumps. These should be the two with highest frequencies. Therefore, **qu** is placed at the middle position of level three and the positions of **abcd** and **rst** are arbitrary. Since **ps** and **lm** have equal frequency their relative positions on level four are arbitrary. While the data structure given in Figure 17 yields an average number of jumps of $\frac{33}{67}$, the optimal assignment requires an average of $\frac{24}{67}$. The B2-optimal tree typically reduces the average number of jumps in decoding a source text by 25–30 percent. There are no disadvantages to the use of the B2-optimal tree for decoding since the only cost is the time it takes the encoder to construct the optimal tree rather than an arbitrary canonical tree, and this cost is small.

6. Summary

Four methods of decoding prefix codes in limited space have been presented. The methods are partitioned into two categories based on the data structuring strategy employed. Method A2 is almost always superior to Method A1. However, the choice among Methods A2, B1, and B2 is less obvious. Parameters of a particular application will influence this decision. Tables comparing time and space requirements of the four methods expose the relevant parameters. The methods we describe define only the decoding phase of a data compression system. The choice of a fixed encoding dictionary is the most critical factor in determining the performance of a system based on our methods. While the representation of the code contributes to the compression ratio attained, most of the compression is achieved by selecting a dictionary that is well-suited to the data being compressed. The size of the dictionary (i.e., number of strings, n) determines the exact time and space requirements of each method. With an advantageous choice of dictionary, our methods can attain compression performance comparable to state-of-the-art techniques such as the UNIX utility *compress*. Defining a dictionary that guarantees good compression is, however, a difficult task. Our methods provide a solution to the problem of decoding in severely limited space without sacrificing compression performance, assuming that preprocessing time to construct the dictionary is available. The methods are described in sufficient detail to allow practitioners to implement them easily.

CHAPTER 4
Space-Limited Context Models of Order 2

Adaptive context modeling has emerged as the most promising new approach to compressing text. While context-modeling algorithms provide very good compression, they suffer from the disadvantages of being relatively slow and requiring large amounts of main memory in which to execute. Algorithm PPMC achieves an average compression ratio of approximately 30 percent. However, PPMC uses 500 Kbytes to represent the model it employs. As memory becomes less expensive and more accessible, machines are increasingly likely to have 500 Kbytes of internal memory available for the task of data compression. There are applications, however, for which it is unreasonable to require this much memory. Examples of these applications include mass-produced special-purpose machines, and software systems in which compressors/decompressors are embedded in larger application programs. Algorithm PPMC has the additional disadvantage of executing at only 2000 characters per second. In this chapter and the next we describe our efforts to improve the practicality of the finite-context modeling concept by streamlining the representation of the model.

The subject of this chapter is order-2 context modeling. Among our order-2 algorithms is one that provides better compression performance than *compress* while using less than 10 percent as much memory. In Section 1 we provide a discussion of the parameters involved in context modeling. In Section 2 we describe previous work on context modeling in limited memory. We discuss the use of self-organizing lists as data compression data structures in Section 3 and describe our order-2 algorithms, which use self-organizing lists, in Section 4. In Section 5 we compare empirical results obtained from implementations of our algorithms with results provided by other researchers. A modification of the algorithm presented in Section 4 that improves both the use of memory and compression performance is the subject of Section 6. In Section 7 we summarize our study of order-2 context modeling.

1. Parameters of Finite-Context Models

Practical context models are not as simple as we described them in Chapter 2. The characterization of an order-i context model as one in which the previous i characters are used to code the current character describes a *pure* order-i model. In fact, pure context models are rarely used, and the more common model is a *blended* order-i model. In a blended order-i model, i is not the order of the only model in use, but the highest order of any model consulted. In a blended model of order i the prediction of the order-i model is combined with predictions of models of lower orders to form a final prediction. Section 1.1 addresses the issue of blending, including the question of which models to blend (i.e., the selection of a maximum context and additional contexts of lower orders). Blending calls for the selection of an escape strategy, which is discussed in Section 1.2. In Section 1.3 we discuss the need for and effects of memory limitations on context models.

1.1. Blending

A blended model of order i is one in which the prediction of the order i model is combined with models of lower orders (e.g., $i-1, i-2, \ldots, 0$). Blending is desirable and essentially unavoidable in an adaptive setting where the model is built from scratch as encoding proceeds. When the first character of a file is read, the model has no history on which to base predictions. Larger contexts become more meaningful as compression proceeds.

In a typical blended order-i model, the number of bits used to code character c will be dictated by the preceding i characters if c has occurred in this particular context before. Otherwise, models of lower order are consulted, generally beginning with order $i-1$, until one of them supplies a prediction. When the context of order i fails to predict the current character, the encoder must emit an *escape* code, a signal to the decoder that the model of lower order is being consulted. The order-0 model may be initialized to provide a prediction for each character so that the process of consulting lower-ordered models terminates. Alternatively, the order-0 model may be used only for characters that have appeared before but are now appearing in a novel context. In this case, a model of order -1 is used for predicting characters when they occur for the first time. The -1 model is initialized so that each of the unused characters is equally likely. When a character occurs in a novel context, this new information is added to the model being constructed. The model of order k consists of a frequency distribution for each k-context (i.e., sequence of k characters) occurring in the file being compressed. Algorithm PPMC is an order-3

blended context model that uses models of order 2, 1, 0 and -1 in addition to the model of order 3.

1.2. Escape Strategy

Each frequency distribution in the blended model must have some frequency allocated to the possibility that the corresponding context does not predict the current character. The code space corresponding to this frequency represents the escape code, a signal that the model does not include a prediction for the current character and that a model of lower order is being consulted. Bell et al. consider several strategies for allocating the escape frequency [BCW90]. There is no theoretical basis for selecting an optimal escape strategy. We adopt the strategy of treating the escape event as if it were an additional symbol in the input alphabet. Like any other character, the frequency of the escape event is the number of times it occurs. Ideally, an escape strategy should take into account the fact that the use of the escape code in a particular context becomes less likely as the number of different characters occurring in that context increases. That is, the escape code indicates a new character (i.e., one that has not occurred in the present context before) and as the number of different characters occurring increases, the probability of a new character decreases.

1.3. Memory Limitations

We will call an order-i context model *full* if, for all $j < i$, every *j-gram* (sequence of j contiguous characters) that occurs in the file being encoded forms an order-j context in the model being constructed. A full model even of order 3 is rare since the space required to store context information for every 3-gram, 2-gram, and single character in the file is prohibitive. There are two obvious ways to impose a memory limit on a finite context model. The first is to monitor its size and to freeze it at the maximum allowable size. When the model is frozen, it can no longer represent characters occurring in novel contexts, but the frequency values already stored in the model can continue to be updated. The second approach is to rebuild the model rather than freeze it. The model can be rebuilt from scratch or from a buffer representing recent history. The use of the buffer may lessen the degradation in compression performance due to rebuilding. On the other hand, the memory set aside for the buffer causes rebuilding to occur earlier. A third approach, which is not strictly speaking a solution to the problem of limited memory, is to monitor the compression ratio as well as the size of the data structure. Rebuilding when the compression ratio has begun to degrade may be more opportune than waiting until it becomes necessary. The PPMC algorithm of Bell et al. uses a full context model of order 3 stored in a tree data

structure that is allowed to grow to 500 Kbytes [BCW90]. The model is rebuilt using the previous 2048 characters when it reaches this limit. The method we describe in Section 4 avoids the problem of exhausting available memory rather than reacting to it.

2. Previous Methods

Langdon and Rissanen describe a algorithm (LR) that uses a subset of the order-1 model [LR83]. Algorithm LR uses a model consisting of z order-1 contexts and an order-0 context (z is a parameter associated with the algorithm and determines its memory requirements). When encoding begins, the order-0 model is used since no characters have yet occurred in any order-1 context. In a full order-1 model, when a character occurs for the first time it becomes an order-1 context. In algorithm LR, only z contexts will be constructed: corresponding to the first z characters that occur at least N times in the text being encoded (N is another parameter of the algorithm). The suggested values $z = 31$ and $N = 50$ provide approximately 50 percent compression with a very modest space requirement and very good speed [BCW90].

Abrahamson presents an order-1 context model with very modest memory requirements. He describes his model as follows:

> "If, for example, in a given text, the probability that the character h follows the character t is higher than that for any other character following a t and the probability of an e following a v is higher than that for any other character following a v, then the same symbol should be used to encode an h following a t as an e following a v. It should be noted that this scheme will also increase the probability of occurrence of the encoded symbol. ... the source message *abracadabra* can be represented by the sequence of symbols *abracadaaaa*. Notice how a b following an a and an r following a b (and also an a following an r) have all been converted into an a, the most frequently occurring source character [A89, p 78]."

We believe a simpler description of Abrahamson's model characterizes it as an order-1 context model that employs a single frequency distribution and that codes symbol y following symbol x as symbol k, where k can be thought of as the position of y on x's list of successors and where successor lists are maintained in frequency count order. Thus we think of *bra* as being coded by 111 rather than *aaa*. The other characters in the string *abracadabra* will also be coded as list positions, but these positions cannot be inferred from the example. While this characterization may not be obvious from the description given above, it becomes clear from the implementation details given in Abrahamson's article [A89].

The data structures in Abrahamson's method consist of two-dimensional arrays *char_to_index*, *index_to_char* and *count*, and one-dimensional frequency and cumulative-frequency arrays. The frequency count array stores in $count[x, y]$ the number of times that character y has appeared in context x (i.e., following character x). The *char_to_index* array is used by the encoder to map characters to frequency values and the *index_to_char* array is used by the decoder to map frequency values to characters. The value of $char\_to\_index[x, y]$ gives the position of y on x's successor list and this position is used to index into the frequency distribution. The single frequency distribution may be thought of as representing the frequencies of occurrence of the various list positions (k values) and this distribution is used for arithmetic coding of the events modeled.

Thus, we recognize that Abrahamson is modifying the basic order-1 model by:

(1) employing a single frequency distribution rather than a distribution for each 1-character context and

(2) employing self-organizing lists to map characters to frequency values.

Abrahamson's model is a *pure* order-1 context model. That is, it is always possible to predict the next character given its predecessor and only order-1 predictions are used. For any pair x, y of successive characters, we code y using the k^{th} frequency value where x is the k^{th} most frequent successor of y. There is intuitive appeal in the use of the frequency count list organizing strategy in Abrahamson's algorithm since the coding technique employed is based on frequency values. On the other hand, the frequency values used are aggregate values. Character y in context x is coded not using $count[x, y]$, but $frequency[k]$ where k is the position of y on the self-organizing list for context x. That is, the frequency used for encoding is not the frequency with which y has occurred after x, but the number of times that position k has been used to encode an event.

3. Self-Organizing Lists and Data Compression

We have seen the move-to-front list organizing strategy used as the heart of a data compression model, algorithm BSTW. Self-organizing lists are also inherent in the most common implementation of arithmetic coding. Maintaining the frequency array in non-increasing order is equivalent to maintaining the corresponding source messages in frequency-count order. Abrahamson employs order-1 context models that are stored in frequency-count order to achieve a simplified order-1 data compression scheme. We have

structure that is allowed to grow to 500 Kbytes [BCW90]. The model is rebuilt using the previous 2048 characters when it reaches this limit. The method we describe in Section 4 avoids the problem of exhausting available memory rather than reacting to it.

2. Previous Methods

Langdon and Rissanen describe a algorithm (LR) that uses a subset of the order-1 model [LR83]. Algorithm LR uses a model consisting of z order-1 contexts and an order-0 context (z is a parameter associated with the algorithm and determines its memory requirements). When encoding begins, the order-0 model is used since no characters have yet occurred in any order-1 context. In a full order-1 model, when a character occurs for the first time it becomes an order-1 context. In algorithm LR, only z contexts will be constructed: corresponding to the first z characters that occur at least N times in the text being encoded (N is another parameter of the algorithm). The suggested values $z = 31$ and $N = 50$ provide approximately 50 percent compression with a very modest space requirement and very good speed [BCW90].

Abrahamson presents an order-1 context model with very modest memory requirements. He describes his model as follows:

> "If, for example, in a given text, the probability that the character h follows the character t is higher than that for any other character following a t and the probability of an e following a v is higher than that for any other character following a v, then the same symbol should be used to encode an h following a t as an e following a v. It should be noted that this scheme will also increase the probability of occurrence of the encoded symbol. ... the source message *abracadabra* can be represented by the sequence of symbols *abracadaaaa*. Notice how a b following an a and an r following a b (and also an a following an r) have all been converted into an a, the most frequently occurring source character [A89, p 78]."

We believe a simpler description of Abrahamson's model characterizes it as an order-1 context model that employs a single frequency distribution and that codes symbol y following symbol x as symbol k, where k can be thought of as the position of y on x's list of successors and where successor lists are maintained in frequency count order. Thus we think of *bra* as being coded by 111 rather than *aaa*. The other characters in the string *abracadabra* will also be coded as list positions, but these positions cannot be inferred from the example. While this characterization may not be obvious from the description given above, it becomes clear from the implementation details given in Abrahamson's article [A89].

The data structures in Abrahamson's method consist of two-dimensional arrays *char_to_index*, *index_to_char* and *count*, and one-dimensional frequency and cumulative-frequency arrays. The frequency count array stores in $count[x, y]$ the number of times that character y has appeared in context x (i.e., following character x). The *char_to_index* array is used by the encoder to map characters to frequency values and the *index_to_char* array is used by the decoder to map frequency values to characters. The value of $char\_to\_index[x, y]$ gives the position of y on x's successor list and this position is used to index into the frequency distribution. The single frequency distribution may be thought of as representing the frequencies of occurrence of the various list positions (k values) and this distribution is used for arithmetic coding of the events modeled.

Thus, we recognize that Abrahamson is modifying the basic order-1 model by:

(1) employing a single frequency distribution rather than a distribution for each 1-character context and

(2) employing self-organizing lists to map characters to frequency values.

Abrahamson's model is a *pure* order-1 context model. That is, it is always possible to predict the next character given its predecessor and only order-1 predictions are used. For any pair x, y of successive characters, we code y using the k^{th} frequency value where x is the k^{th} most frequent successor of y. There is intuitive appeal in the use of the frequency count list organizing strategy in Abrahamson's algorithm since the coding technique employed is based on frequency values. On the other hand, the frequency values used are aggregate values. Character y in context x is coded not using $count[x, y]$, but $frequency[k]$ where k is the position of y on the self-organizing list for context x. That is, the frequency used for encoding is not the frequency with which y has occurred after x, but the number of times that position k has been used to encode an event.

3. Self-Organizing Lists and Data Compression

We have seen the move-to-front list organizing strategy used as the heart of a data compression model, algorithm BSTW. Self-organizing lists are also inherent in the most common implementation of arithmetic coding. Maintaining the frequency array in non-increasing order is equivalent to maintaining the corresponding source messages in frequency-count order. Abrahamson employs order-1 context models that are stored in frequency-count order to achieve a simplified order-1 data compression scheme. We have

structure that is allowed to grow to 500 Kbytes [BCW90]. The model is rebuilt using the previous 2048 characters when it reaches this limit. The method we describe in Section 4 avoids the problem of exhausting available memory rather than reacting to it.

2. Previous Methods

Langdon and Rissanen describe a algorithm (LR) that uses a subset of the order-1 model [LR83]. Algorithm LR uses a model consisting of z order-1 contexts and an order-0 context (z is a parameter associated with the algorithm and determines its memory requirements). When encoding begins, the order-0 model is used since no characters have yet occurred in any order-1 context. In a full order-1 model, when a character occurs for the first time it becomes an order-1 context. In algorithm LR, only z contexts will be constructed: corresponding to the first z characters that occur at least N times in the text being encoded (N is another parameter of the algorithm). The suggested values $z = 31$ and $N = 50$ provide approximately 50 percent compression with a very modest space requirement and very good speed [BCW90].

Abrahamson presents an order-1 context model with very modest memory requirements. He describes his model as follows:

> "If, for example, in a given text, the probability that the character h follows the character t is higher than that for any other character following a t and the probability of an e following a v is higher than that for any other character following a v, then the same symbol should be used to encode an h following a t as an e following a v. It should be noted that this scheme will also increase the probability of occurrence of the encoded symbol. ... the source message *abracadabra* can be represented by the sequence of symbols *abracadaaaa*. Notice how a b following an a and an r following a b (and also an a following an r) have all been converted into an a, the most frequently occurring source character [A89, p 78]."

We believe a simpler description of Abrahamson's model characterizes it as an order-1 context model that employs a single frequency distribution and that codes symbol y following symbol x as symbol k, where k can be thought of as the position of y on x's list of successors and where successor lists are maintained in frequency count order. Thus we think of *bra* as being coded by 111 rather than *aaa*. The other characters in the string *abracadabra* will also be coded as list positions, but these positions cannot be inferred from the example. While this characterization may not be obvious from the description given above, it becomes clear from the implementation details given in Abrahamson's article [A89].

The data structures in Abrahamson's method consist of two-dimensional arrays *char_to_index*, *index_to_char* and *count*, and one-dimensional frequency and cumulative-frequency arrays. The frequency count array stores in $count[x, y]$ the number of times that character y has appeared in context x (i.e., following character x). The *char_to_index* array is used by the encoder to map characters to frequency values and the *index_to_char* array is used by the decoder to map frequency values to characters. The value of $char\_to\_index[x, y]$ gives the position of y on x's successor list and this position is used to index into the frequency distribution. The single frequency distribution may be thought of as representing the frequencies of occurrence of the various list positions (k values) and this distribution is used for arithmetic coding of the events modeled.

Thus, we recognize that Abrahamson is modifying the basic order-1 model by:

(1) employing a single frequency distribution rather than a distribution for each 1-character context and

(2) employing self-organizing lists to map characters to frequency values.

Abrahamson's model is a *pure* order-1 context model. That is, it is always possible to predict the next character given its predecessor and only order-1 predictions are used. For any pair x, y of successive characters, we code y using the k^{th} frequency value where x is the k^{th} most frequent successor of y. There is intuitive appeal in the use of the frequency count list organizing strategy in Abrahamson's algorithm since the coding technique employed is based on frequency values. On the other hand, the frequency values used are aggregate values. Character y in context x is coded not using $count[x, y]$, but $frequency[k]$ where k is the position of y on the self-organizing list for context x. That is, the frequency used for encoding is not the frequency with which y has occurred after x, but the number of times that position k has been used to encode an event.

3. Self-Organizing Lists and Data Compression

We have seen the move-to-front list organizing strategy used as the heart of a data compression model, algorithm BSTW. Self-organizing lists are also inherent in the most common implementation of arithmetic coding. Maintaining the frequency array in non-increasing order is equivalent to maintaining the corresponding source messages in frequency-count order. Abrahamson employs order-1 context models that are stored in frequency-count order to achieve a simplified order-1 data compression scheme. We have

investigated the performance of other self-organizing list strategies in connection with Abrahamson's model (for a survey of list organizing strategies, see [HH85]). We tested the performance of move-to-front, transpose, and move-p%-of-the-way-to-front for $p = 33$, 50, 67, 70, and 75. For most files we tested, frequency count provided the best compression ratios, but the differences in performance were not dramatic. Our results agree with research by Horspool and Cormack in which a variety of list organizing strategies are used in connection with an order-0 context model based on words rather than characters. They also report no significant performance differences among list organizing methods [HC87]. The use of transpose or move-to-front obviates the need for frequency counts in Abrahamson's algorithm and reduces the memory requirement from 200 Kbytes to 68 Kbytes when $n = 256$ (n is the size of the input alphabet) without significantly degrading compression performance.

When used to encode text files (where the alphabet size is typically in the range 90–128), Abrahamson's algorithm provides a speed advantage over the context-model-based algorithm PPMC. However, the compression performance, approximately 54 percent, compares poorly with that provided by PPMC and *compress*. The space requirement of Abrahamson's algorithm is modest compared to algorithm PPMC and *compress*, but it is greater than that of algorithm FG for alphabets of size greater than 200. Using the transpose list organizing strategy instead of frequency count improves the space requirements of Abrahamson's algorithm, but provides the same mediocre compression performance. And, as we have noted, Abrahamson's method does not extend naturally beyond order-1 context modeling. Using a single frequency distribution would provide some memory reduction, but to maintain a self-organizing list of size n (where $n = 128$ or 256) for each two-character context is prohibitive. Our order-2 model uses self-organizing lists of size s, where $s \ll n$, for each two-character context. We provide a complete description of our algorithm, based on an order-2 blended context model, in the next section.

4. Fast Order-2 Context Models in Limited Memory

The algorithm we describe in this section employs a blended order-2 context model. It can be implemented so as to provide compression performance that is better than that provided by *compress* and much better than that provided by Abrahamson's method, using far less space than either of these systems. In Section 4.1 we describe the method of blending we use, and in Section 4.2 we provide more detail on our use of self-organizing lists. In Section 4.3 we describe our use of arithmetic coding, and Section 4.4 presents our

escape strategy. We discuss the memory requirement of our algorithm and its execution speed in Section 4.5.

4.1. Blending Strategy

One of the ways in which we conserve on both memory and execution time is by blending only models of orders 2 and 0, rather than orders 2, 1, 0, and -1. Thus we refer to our model as an order-2-and-0 context model. We have experimented with order-2-and-1 and order-2-1-and-0 models. The order-2-and-1 model did not provide satisfactory compression performance and the order-2-1-and-0 model produces compression results that are very close to those of our order-2-and-0 algorithm. The order-2-and-0 model allows faster encoding and decoding since it consults at most two contexts per character. We provide more details on the models of orders 2 and 0 and how they are blended in the next section.

4.2. Self-Organizing Lists

In our order-2-and-0 model, we maintain a self-organizing list of size s for each two-character context. We encode z when it occurs in context xy by *event* k if z is in position k of list xy. When z does not appear on list xy we encode z itself. The order-0 part of the model consists of frequency values for n characters. Encoding entails mapping the event (k or z) to a frequency and employing an arithmetic coder. To complete the description of the model, we need to specify a list organizing strategy and the method of maintaining frequencies. The frequency count list organizing strategy is inappropriate because of the large number of counts required. We employ the transpose strategy because it provides faster update than move-to-front.

When character z occurs in context xy and z appears on the context list for xy, the list is updated using the transpose strategy. If z does not appear on the xy list, it is added. If the size of list xy is less than s ($size < s$), the item currently in position $size$ moves into position $size + 1$ and z is stored in position $size$. If the list is full when z is to be added, z will replace the last item. An obvious disadvantage to fixing the size of the order-2 context lists is that the lists are likely to be too short for some contexts and too long for others. When an order-2 list (say, list xy) has s items and a new character z occurs in context xy, we delete the bottom item (call it t) from the list and add z. Context xy no longer predicts t. This does not affect the correctness of our algorithm. When t occurs again in context xy it will be predicted by the order-0 model. The fact that encoder and decoder maintain identical models ensures correctness. In addition, the rationale behind the use of

self-organizing lists is that we expect to have the s most common successors on the list at any point in time. As characteristics of the file change, successors that become common replace those that fall into disuse. The method of maintaining frequencies and using them to encode is described in the next section.

4.3. Arithmetic Coding

In order to conserve memory we do not use a frequency distribution for each context. Instead, we maintain a frequency value for each feasible event. Since there are $s+1$ values of k (the s list positions and the escape code) and $n+1$ values for z (the n characters of the alphabet and an end-of-file character), the number of feasible events is $s+n+2$. We can maintain the frequency values either as a single distribution or as two distributions, an order-2 distribution to which list positions are mapped and an order-0 distribution to which characters are mapped. Our experiments indicate that the two-distribution model is slightly superior. When z occurs in context xy we use the two frequency distributions in the following way: if list xy exists and z occupies position k, we encode k using the order-2 distribution. If list xy exists but does not contain z, we encode an escape code (using the order-2 distribution) as a signal to the decoder that an order-0 prediction (and the order-0 frequency distribution) is to be used, and then encode the character z. When list xy has not been created yet, the decoder knows this and no escape code is necessary; we simply encode z using the order-0 distribution.

4.4. Escape Strategy

The escape code must be chosen so that the decoder recognizes it as a signal rather than a legitimate list position. If viewed as an additional list position, there are two reasonable choices for the value of the escape. One choice is to use the value $s+1$, as it will never represent a list position. The second choice is to use the value $size+1$, where $size$ is the current size of list xy (and ranges from 1 to s). In the first case, the escape code is the same for every context and all of the counts for the escape code accrue to a single frequency value while in the second case, the value of the escape code depends on the context and generates counts that accrue to multiple frequency values. The two escape strategies produce similar compression results. The algorithm we describe here uses the first alternative.

In Section 4.2 we specified the way in which the self-organizing lists are updated. The frequency distributions are updated in much the same way. That is, a frequency distribution is updated when it is used. Thus, when list xy exists, the order-2 distribution

is updated after it is used to encode either a list position or an escape. The order-0 distribution is used and updated each time context xy fails to predict z.

4.5. Memory Requirement and Execution Speed

The data stored for our method includes frequency and cumulative frequency lists of size $s+2$ (for order 2) and $n+2$ (for order 0), and *pos_to_freq* and *freq_to_pos* arrays of size $s+1$ and n, as well as the self-organizing lists of size s. The *pos_to_freq* and *freq_to_pos* arrays play the role of Abrahamson's *char_to_index* and *index_to_char* arrays, mapping list positions to frequencies in the order-2 context and characters to frequencies in the order-0 context. When the self-organizing lists are implemented as arrays, the total memory requirement of our method is $n^2(s+1)+5(n+1)+3(s+1)+6$ bytes. With an s value as low as 2, our method is faster than Abrahamson's and provides better compression with less storage required. Based on empirical data, $s=7$ provides the best average compression over a suite of test files. With $s=7$ we use approximately three times as much memory as Abrahamson's method but achieve compression that is 20 percent better on average (3.16 bits per character as opposed to 3.94) and in slightly less execution time. Our method also provides better compression than *compress* (approximately 15 percent better with $s=7$) using essentially the same memory requirement for $n=256$ and far less for $n=128$.

Using dynamic memory allocation to implement the self-organizing lists results in a much more efficient use of space. We allocate an array of n^2 pointers to potential lists, and allocate space for list xy only if xy occurs in the text being compressed. The memory requirement becomes $n^2+u(s+1)+5(s+n+2)+3$ bytes, where u represents the number of distinct character pairs occurring in the text. In our suite of test files, the maximum value of u was 4721. This value was encountered in file **windows**, a 0.69 megabyte file of messages extracted from the bulletin board *comp.windows.x*. Even in this worst case, the dynamic-memory version of the order-2-and-0 algorithm results in a 95 Kbyte space savings over Abrahamson's method (when both methods use $k=256$ and with $s=7$, our space requirement is $\approx$ 104 Kbytes and Abrahamson's $\approx$ 199 Kbytes). The compression performance is, of course, the same as that provided by an array-based implementation.

The dynamic-memory implementation is slightly slower than the static version due to overhead incurred by dynamic allocation, but this algorithm is still faster than Abrahamson's algorithm. C-language versions of Abrahamson's algorithm and our dynamic-memory implementation compress approximately 1900 and 3000 characters per second, respectively. We estimate that when our implementation is optimized, its speed

will be competitive with that of algorithm FG. The execution time of our algorithm is determined by the size of the input file, the size of the output file, and the lengths of the self-organizing lists. For each input character we consult and update one or both models, and use and update the corresponding code(s). The time contribution due to the order-2 model consists of the time required to search for the current character on an order-2 list and the time required to update the order-2 list and corresponding frequency distribution. The time to search and update the model is limited by the current size of the self-organizing list (which is in turn bounded by the maximum list size, s). We expect the frequently-occurring characters to be near the front of the context lists, so that the average time spent in manipulating the order-2 model should be much less than the maximum list length, s. When the order-2 model does not supply a prediction, the order-0 model must be consulted. Consulting the order-0 model requires very little time since the mapping *pos_to_freq* provides immediate access to the frequency value corresponding to the current character. Updating the order-0 model involves maintaining the characters in frequency-count order, so that a single update could require n operations. However, the frequency-count strategy maintains frequently-occurring characters near the front of the list so that the average cost is again much less than the maximum possible.

When an order-2 list contains fewer than s items, we are subject to the criticism that we are not putting our memory resources where we need them. In fact, fixing the number of successors represents a tradeoff of the ability to predict any character against the ability to predict quickly while using a reasonable amount of space. Fixing the number of successors suggests the use of an array data structure rather than a linked structure; thus we avoid the space required for links and the time involved in creating and updating linked nodes. The links in a linked structure may also be viewed as consuming memory without directly representing information needed for prediction. Another disadvantage of the linked structure is that it is more difficult to control its growth. In algorithm PPMC, the trie is simply allowed to grow until it reaches a limit and then is discarded and rebuilt. Rebuilding can result in loss of prediction accuracy. When rebuilding takes place, all of the information constructed from the prefix of the file is lost. By contrast, our model loses only the ability to predict certain successors in certain contexts, and only when they have ceased to occur frequently. Finally, we must keep in mind that a dynamic data compression system attempts to "hit a moving target". When characteristics of the file being compressed change, it may be advantageous to lose some of the data collected in

File type	Order 2-and-0	Unix Compress	Unix Compact	Abrahamson's Order-1
bboard	44.88	47.69	69.91	51.61
doc	39.57	42.85	57.16	48.98
TEX	39.99	43.09	61.02	50.14
source	33.98	42.69	61.17	46.63
non-text	49.33	55.93	74.42	57.08
all	39.53	45.36	61.87	49.21

Table 4
Compression ratio by category (expressed as percentage)

compressing the early part of the file. Unfortunately, we can only make an intelligent guess at what information to collect and when to discard it.

5. Experimental Results

We compare the performance of the order-2-and-0 method to that of *compress*, *compact*, and Abrahamson's method on a suite of 34 files selected to include a variety of file types and sizes. Since *compress* and *compact* are available under UNIX and source code for Abrahamson's method appears in [A89], we are able to run each of these methods against our test suite. Where possible, we include files used by other researchers to compare with competing compression algorithms. The files we use can be grouped into categories: *bboard* files consisting of electronic bulletin board entries, *doc* files of on-line program documentation/user's manuals, TEX-formatted versions of technical papers, *source* files in C and Pascal, *non-text* files including a *dvi* file and a binary file, and *miscellaneous* file types. The miscellaneous file category includes files **alphabet** (enough copies of the 26-letter alphabet to fill out 100,000 characters) and **skewstat** (10,000 copies of the string *aaaabaaaac*) described by Witten et al. [WNC87] and the UNIX dictionary **/usr/dict/words** described by Williams [W88]. Table 4 presents a performance comparison for the order-2-and-0 algorithm with $s = 7$ and with space requirement ≈ 104 Kbytes. Data reported are average compression ratios by category and overall. The order-2-and-0 algorithm outperforms the competing methods in every category. The performance of *compact* is clearly unacceptable.

File	Original Size	Order 2-and-0	Unix Compress	Abrahamson's Order-1
/usr/dict/words	201089	38.09	51.10	49.33
fcsh	77844	37.27	38.10	44.30
ocsh	118784	55.71	65.35	62.68
comp20	578	64.53	83.56	80.45
comp50	1234	55.02	68.80	66.94
comp100	2292	46.64	59.82	58.38
comp200	4877	46.24	58.48	58.60
comp500	13314	44.76	54.35	55.88
compress	35382	40.26	47.67	51.81

Table 5

Performance on selected files (compression ratio in percent form)

In Table 5 we display compression results for some specific files. These are: **/usr/dict/words** described above; **fcsh**, the formatted manual entry for the *csh* command in UNIX; **ocsh**, the object code for the *csh* command; **compress20** through **compress500**; and **compress**. **Compress** is the C source code for the UNIX utility *compress* and **compress20** contains the first 20 lines of **compress**. Original file sizes are listed in column two.

Williams reports results on **/usr/dict/words** and the various **compress** files [W88]. The values he gives for original file sizes are slightly different from ours since local copies of the files contain minor differences. Williams' dynamic-history compression technique achieves a compression ratio of 58.3 on **/usr/dict/words** and ratios of 69.9, 57.3, 45.4, 49.2, 40.1, and 42.24 on the versions of the *compress* source. Williams' motivation in considering subsets of the *compress* source was to emphasize the fact that his model 'learns' the characteristics of a file as it compresses. Thus, a larger file provides more opportunity for learning and greater compression is achieved. Any dynamic data compression scheme learns characteristics of a source as compression proceeds. Compression performance improves with file size to the point at which the limit on available memory is reached. When the algorithm can no longer store new information, performance may degrade. The

compress files and the *source* category in Table 4 demonstrate that the order-2-and-0 method performs particularly well on source program files. Cormack and Horspool report results for files **fcsh** and **ocsh** [CH87]. The values for original file size differ from ours substantially, so comparisons are unreliable. Cormack and Horspool report that an order-4 context model achieves compression ratios of 26.5 and 69.4 on **fcsh** and **ocsh** respectively and that a dynamic Markov model provides ratios of 27.2 and 54.8. Our results for **ocsh** compare favorably to theirs. In comparing ratios, however, we must keep in mind that the files may be quite different and, more importantly, that the models they discuss have unlimited memory requirements.

Cleary and Witten report results for a blended order-2 model that uses orders 2, 1, 0, and -1 [CW84]. For text files their algorithm produces an average of 3.31 bits per character (bpc). We can compare this with 3.17 bpc for document files and 3.20 for TEX files, obtained by multiplying the compression ratios in Table 4 by 8 (the input files were in 8-bit ASCII form). Cleary and Witten also report compressing source files to 2.92 bpc while our *source* category is compressed to an average of 2.72 bpc. While we compress the object file **ocsh** to 4.46 bpc, Cleary and Witten report a 4.93 figure for a binary file. The implementation of the order-2 model employed by Cleary and Witten is similar to that of PPMC, so the space requirements are much greater than those of our order-2-and-0 model.

6. Using Hashing to Improve Memory Use

We have described an algorithm that allocates n^2 self-organizing lists of size s and another that uses dynamic memory to allocate lists of size s only when they are needed. The second algorithm, however, statically allocates n^2 pointers, one for each of the n^2 possible contexts. In this section we describe an order-2-and-0 strategy that uses hashing rather than dynamic memory. This algorithm employs a hash table into which all n^2 contexts are hashed. Each hash table entry is a self-organizing list of size s. An implementation of this strategy provides better average compression than the earlier methods and requires much less memory.

Encoding and decoding proceed as in the earlier algorithms. When z occurs in context xy and no xy list exists we encode z using the order-0 frequency distribution. When an xy list exists but does not contain z, we emit an escape code and then code z using the order-0 distribution. When z is contained on the list for xy we code its position. An obvious disadvantage of the use of hashing is the possibility of collision. If two or more contexts (say xy and ab) hash to the same table position, the lists for these contexts are

coalesced into a single self-organizing list used to represent both contexts. We can view this as xy's successors vying with those of ab for position on the list. Intuitively, it would seem that our predictions are more accurate when xy and ab are represented by separate lists. However, we repeat our admonitions on the unreliability of intuition in compressing text. It is possible that when we expect it least the characteristics of our file change and our good statistics become bad. Thus, we can be optimistic and hope that coalescing two lists will a) happen infrequently, b) not degrade performance, or c) will actually improve performance.

Hash conflicts have no impact on the correctness of the approach; they may, however, impact compression performance. We mitigate the negative effects of hashing in three ways. First, we select the hash function so as to minimize the occurrence of collisions. Second, we use double hashing to resolve collisions. In order to resolve collisions, we must be able to detect them. We detect collisions by storing with the self-organizing list an indication of the context to which it corresponds. When context xy hashes to position h but the check value at position h does not correspond to context xy, we know that we have collision. In order to maintain reasonable running time we perform only a small number of probes (four in the implementation we describe here). If the short probe sequence does not resolve the hash conflict, we allow the two lists to coalesce.

The third way in which we minimize the negative effects of hashing is to use some of the space gained by eliminating n^2 pointers to provide $m > 1$ order-2 frequency distributions. The value of m is is significantly smaller than the size of the hash table (H) so that we are coalescing H/m lists into each frequency distribution. Thus the cost is less than that of providing a frequency distribution for each context while compression results are better than those achieved when we use a single frequency distribution for all lists. The disadvantages of coalescing frequency distributions are the same as those of coalescing lists except that coalescing lists is likely to cause loss of the ability to predict some characters (since two contexts now have $s3$ list positions between them instead of $s3$ each), and limiting the number of frequency distributions does not cause this problem. Because the number of frequency distributions is very small relative to the number of contexts of order 2 we consider hash collisions to be inevitable and do not attempt to resolve them.

File/Type	Order 2-and-0 (Dynamic)	Order 2-and-0 (Hashing)
bboard	44.88	43.98
doc	39.57	38.85
TEX	39.99	39.03
source	33.98	33.80
non-text	49.33	48.69
all	41.09	38.90
/usr/dict/words	38.09	36.48
fcsh	37.27	35.79
ocsh	55.71	56.78
comp20	64.53	67.30
comp50	55.02	55.75
comp100	46.64	46.34
comp200	46.24	46.16
comp500	44.76	44.43
compress	40.26	39.54

Table 6

Compression ratios — dynamic memory and hashing

An implementation of the hash-based algorithm with $H = 4800$, $m = 70$, $s = 7$, and $n = 256$ provides approximately 6 percent more compression than the order-2-and-0 algorithm described above and uses only 45 Kbytes of memory (less than half of the requirement of the method of Section 4). We provide empirical comparisons with the pointer-based algorithm in Table 6. The use of hashing provides improved compression performance overall. The only files on which the hash-based algorithm is inferior are very short files, such as **comp20** and **comp50**, and the object code file **ocsh**.

7. Summary

We present an order-2-and-0 finite context model that provides compression performance better than that of the state-of-the-art UNIX utility *compress* and has memory requirements far more modest than those of *compress*. Our algorithm provides improved

compression performance on files of many types and performs particularly well on program source codes. The use of hashing described in Section 6 results in a technique for streamlining context models that may be applied to models of any order. The sizes of the hash tables used to store self-organizing lists and frequency distributions are parameters of the approach, the values of which determine space efficiency and impact compression performance. In Chapter 5 we discuss our research on order-3 context modeling in limited space.

CHAPTER 5
Space-Limited Context Models of Order 3

1. Fast Order-3 Context Models in Limited Memory

In this chapter we extend our work on context modeling in limited memory to context models of order 3. The use of hashing to store context information permits the extension of the strategy developed in Chapter 4 to blended models of arbitrary order. The primary problem in designing an order-3 algorithm with modest memory requirements is that of deciding which lower-ordered models to blend with the order-3 model. We concentrate our discussion on the blended order-3 context model that gives the best overall results. Our algorithm has a much more modest memory requirement than competing algorithms FG and PPMC and provides compression performance that is superior on average to that provided by algorithm FG. In addition, its speed is superior to that of algorithm PPMC. When tuned, we expect encode speed comparable to that of the faster algorithm FG. In Section 1.1 we discuss the method of blending we employ, and in Section 1.2 the data structures used. Section 1.3 details the way in which the predictions supplied by our model are coded, and in Section 1.4 we discuss the memory requirements and execution speed of our order-3 algorithm.

1.1. Blending Strategy

The best algorithm in our family is based on an order-3-1-and-0 context model. That is, we construct a prediction for the character being encoded by blending predictions based on the previous three characters, the previous character, and unconditioned character counts. We considered order-3-and-0 models and order-3-2-and-0 models as well as the order-3-1-and-0 approach that we describe here. The addition of order-2 context information to the order-3-and-0 model generally did not improve compression performance, while the addition of contexts of order 1 does provide significantly better results. Eliminating some of the models of lower order contributes to both the decreased memory requirement and increased speed of our methods. Thus we limited the total number of contexts to be blended to three, and did not consider models that blended orders 3, 2, 1, and 0, for example. In the next section we describe the way in which we store context information.

1.2. Data Structures

We use self-organizing lists to maintain the order-3 and order-1 context information. As in the order-2-and-0 model, we employ the transpose list organizing strategy. The order-3 context information is stored in two hash tables, a hash table $H3$ of size $h3$ whose elements are self-organizing lists of size $s3$, and a hash table $F3$ containing $f3$ frequency distributions. Thus, each trigram (i.e., context of order 3) appearing in the file being compressed is mapped to a position in the hash table $H3$, where a list of $s3$ successor characters is stored. A second hash function maps the trigram to a position in the hash table $F3$ that stores the frequency distribution corresponding to the $s3$ successor characters. Since there are only n order-1 lists (where n is the size of the character set), hashing is not used to store the self-organizing lists of order 1. We maintain a list of $s1$ successors for each single character (context of order 1). However, we maintain just $f1$ (where $f1 < n$) frequency distributions for the collection of order-1 lists. Thus while the order-3 model is a essentially a two-level hashing scheme, where a context hashes first to a position in the table of self-organizing lists and then to a smaller table of frequency distributions, the order-1 model employs just one level of hashing, mapping the n contexts to $f1$ frequency distributions. The order-0 data consists of n frequency, cumulative frequency, and map values; one for each of the n symbols of our alphabet.

1.3. Coding the Model

The models of order 3, 1, and 0 are used to form a prediction of the current character in much the same way as we used them in the order-2-and-0 algorithm of Chapter 4. We use arithmetic coding to map our predictions to a bit stream. We encode character z occurring in context wxy by event k if z occurs in position k of the list for context wxy. If z does not appear on wxy's list, we code an escape and consult the list for the order-1 context y. An order-3 frequency distribution is used to code either k or *escape*. When the order-1 model is consulted, an order-1 frequency distribution is used to code either j (if z occurs in position j of list y) or or an escape code. When neither context wxy nor context y predicts z we follow the two escape codes with an order-0 prediction (i.e., we code the character itself). If the list for context wxy (likewise context y) is empty, the corresponding escape code is not necessary. The decoder maintains the same model of the data as the encoder and knows that since context wxy has never occurred before it cannot supply a prediction.

The escape codes are represented as list positions $s3+1$ and $s1+1$, respectively. The update strategy we use is essentially the same one we described in Section 4.4 of Chapter 4.

The essence of the strategy is that lists and frequency distributions are updated only when they contribute to the prediction of the current character, z. If list wxy exists, we update it using the transpose heuristic. If no wxy list exists one will be created. If context wxy does not predict z, then the y list is updated using the transpose method. If list y is not used in the prediction, it is not updated.

We also update each frequency distribution used in the prediction. When list wxy exists, the wxy frequency distribution is updated after it is used to encode either a list position or an escape. When context wxy does not predict and list y exists, the y frequency distribution is updated. The order-0 frequency distribution is updated whenever the character itself is coded.

1.4. Memory Requirement and Execution Speed

Our order-3-1-and-0 algorithm is in fact a family of algorithms where each algorithm in the family corresponds to a different set of values for the parameters $s3$, $h3$, $f3$, $s1$, and $f1$. The space requirements, speed, and compression performance of a particular algorithm depend on the values of these parameters. We report results in Section 2 for an algorithm that executes in 100 Kbytes of memory and encodes and decodes approximately 2800 cps. Bell et al. report compression speeds for competing algorithms running on a 1-MIP VAX 11/780 [BCW90]. In order to provide a meaningful comparison of running times, we execute on our research machine the order-3-1-and-0 algorithm and the version of *compress* used by Bell et al. Using the execution time of *compress* as a baseline, we adjust the running time of our algorithm to reflect the difference in machines. While this approach is obviously imperfect, it provides a reasonable basis for comparison. Our programs are part of a research testbed and have not been optimized for speed. We believe that with some attention to optimization they can be tuned to compress at approximately the same rate as algorithm FG.

2. Experimental Results

We compare the performance of our order-3-1-and-0 model to that of *compress* and the 45-Kbyte method of Chapter 4 on the suite of files described in Chapter 4. In Table 7 we display compression ratios by category and for selected files. The order-3-1-and-0 model used here has parameter settings: $s3 = 3$, $h3 = 12000$, $f3 = 900$, $s1 = 20$, $f1 = 256$ and uses less than 100 Kbytes of internal memory. Original file sizes are listed in column two. The performance of the order-3-1-and-0 model is significantly better than that of the order-2-and-0 method, which provides dramatic gains over the state-of-the-art *compress*.

File	Original Size	Order 3-1-and-0	Order 2-and-0	Unix Compress
bboard		38.69	43.98	47.69
doc		34.04	38.85	42.85
TEX		32.12	39.03	43.09
source		29.46	33.80	42.69
non-text		44.74	48.69	55.93
all		34.78	38.90	45.36
/usr/dict/words	201089	35.12	36.48	51.10
fcsh	77844	29.72	35.79	38.10
ocsh	118784	56.08	56.78	65.35
comp20	578	68.34	67.30	83.56
comp50	1234	54.94	55.75	68.80
comp100	2292	44.37	46.34	59.82
comp200	4877	42.65	46.16	58.48
comp500	13314	40.12	44.43	54.35
compress	35382	35.07	39.54	47.67

Table 7

Comparison to *compress* and order-2-and-0

The order-3-1-and-0 algorithm reduces a file to an average of 35 percent of its original size, while the order-2-and-0 method reduces files to 39 percent of original size on average, and *compress* leaves 55 percent of the original size.

We compare the performance of our order-3-1-and-0 model to the performance of algorithms BSTW, FG, and PPMC in Table 8. The collection of files for which we report results is the corpus used by Bell et al. to measure the performance of a collection of data compression methods [BCW90]. The files again represent a variety of sizes and types: **obj1** and **obj2** are executable files for two different machines, **geo** is a file of 32-bit numbers representing seismic data, **pic** is a bit map of a black and white facsimile picture. The remaining files are ASCII files of various types: **progc** is the source of **compress**, **progp** and **progl** are source files of LISP and Pascal programs, respectively. The data for algorithms BSTW, FG, and PPMC is taken from [BCW90]. The compression performance

References

[A89] Abrahamson, D. M. An adaptive dependency source model for data compression. *Commun. ACM 32*, 1 (Jan., 1989), 77–83.

[AF87] Apostolico, A. and Fraenkel, A. S. Robust transmission of unbounded strings using Fibonacci representations. *IEEE Trans. Inf. Theory 33*, 2 (Mar., 1987), 238–245.

[BCW90] Bell, T., Cleary, J. G., and Witten, I. H. *Text Compression*, Prentice-Hall, Englewood Cliffs, N.J., 1990.

[BM89] Bell, T. and Moffat, A. A note on the DMC data compression scheme. *Comput. J. 32*, 1 (Feb., 1989), 16–20.

[BSTW86] Bentley, J. L., Sleator, D. D., Tarjan, R. E., and Wei, V. K. A locally adaptive data compression scheme. *Commun. ACM 29*, 4 (Apr., 1986), 320–330.

[CFKP86] Choueka, Y., Fraenkel, A. S., Klein, S. T., and Perl, Y. Huffman coding without bit-manipulation. Tech. Rep. CS86-05. Dept. of Applied Mathematics, The Weizmann Institute of Science, Rehovot, Israel (1986).

[CW84] Cleary, J. G. and Witten, I. H. Data compression using adaptive coding and partial string matching. *IEEE Trans. Comm. 32*, 4 (Apr., 1984), 396–402.

[C73] Connell, J. B. A Huffman-Shannon-Fano code. *Proc. IEEE 61*, 7 (July, 1973), 1046–1047.

[CH87] Cormack, G. V. and Horspool, R. N. S. Data compression using dynamic Markov modeling. *Comput. J. 30*, 6 (Dec., 1987), 541–550.

[E75] Elias, P. Universal codeword sets and representations of the integers. *IEEE Trans. Inf. Theory 21*, 2 (Mar., 1975), 194–203.

[E87] Elias, P. Interval and recency rank source coding: two on-line adaptive variable-length schemes. *IEEE Trans. Inf. Theory 33*, 1 (Jan., 1987), 3–10.

[F73] Faller, N. An adaptive system for data compression. In *Record of the 7th Asilomar Conf. on Circuits, Systems and Computers*, Pacific Grove, Calif., 1973, pp. 593–597.

[FG89] Fiala, E. R. and Greene, D. H. Data compression with finite windows. *Commun. ACM 32*, 4 (Apr., 1989), 490–505.

[G78] GALLAGER, R. G. Variations on a theme by Huffman. *IEEE Trans. Inf. Theory 24*, 6 (Nov., 1978), 668–674.

[H79] HANKAMER, M. A modified Huffman procedure with reduced memory requirements. *IEEE Trans. Commun. 27*, 6 (June, 1979), 930–932.

[HH85] HESTER, J. H. AND HIRSCHBERG, D. S. Self-organizing linear search. *ACM Comput. Surv. 17*, 3 (Sept., 1985), 295–311.

[HC86] HORSPOOL, R. N. AND CORMACK, G. V. Dynamic Markov modelling — a prediction technique. *Proc. International Conference on the System Sciences*, Honolulu, HA. (Jan., 1986).

[HC87] HORSPOOL, R. N. AND CORMACK, G. V. A locally adaptive data compression scheme. *Commun. ACM 16*, 2 (Sept., 1987), 792–794.

[H52] HUFFMAN, D. A. A method for the construction of minimum- redundancy codes. *Proc. IRE 40*, 9 (Sept., 1952), 1098–1101.

[K85] KNUTH, D. E. Dynamic Huffman coding. *J. Algorithms 6*, 2 (June, 1985), 163–180.

[LR83] LANGDON, G. G. AND RISSANEN, J. J. A double-adaptive file compression algorithm. *IEEE Trans. Comm. 31*, 11 (Nov., 1983), 1253–1255.

[LH87] LELEWER, D. A. AND HIRSCHBERG, D. S. Data compression. *ACM Comput. Surv. 19*, 3 (Sept., 1987), 261–296.

[Mo89] MOFFAT, A. Word-based text compression. *Software - Practice and Experience 19*, 2 (Feb., 1989), 185–198.

[R87] RYABKO, B. Y. A locally adaptive data compression scheme. *Commun. ACM 16*, 2 (Sept., 1987), p. 792.

[S64] SCHWARTZ, E. S. An optimum encoding with minimum longest code and total number of digits. *Inform. Contr. 7*, 1 (Mar., 1964), 37–44.

[SK64] SCHWARTZ, E. S. AND KALLICK, B. Generating a canonical prefix encoding.. *Commun. ACM 7*, 3 (Mar., 1964), 166–169.

[SW49] SHANNON, C. E. AND WEAVER, W. *The Mathematical Theory of Communication*, University of Illinois Press, Urbana, Ill., 1949.

[Si88] SIEMINSKI, A. Fast decoding of the Huffman codes. *Inf. Process. Lett. 26*, 5 (May, 1988), 237–241.

[St88] STORER, J. A. *Data Compression: Methods and Theory*, Computer Science Press, Rockville, Md., 1988.

[T87] TANAKA, H. Data structure of Huffman codes and its application to efficient encoding and decoding. *IEEE Trans. Inf. Theory 33*, 1 (Jan., 1987), 154–156.

[V87] VITTER, J. S. Design and analysis of dynamic Huffman codes. *J. ACM 34*, 4 (Oct., 1987), 825–845.

[W84] WELCH, T. A. A technique for high-performance data compression. *Computer 17*, 6 (June, 1984), 8–19.

[W88] WILLIAMS, R. N. Dynamic-history predictive compression. *Information Systems 13*, 1 (Jan., 1988), 129–140.

[WNC87] WITTEN, I. H., NEAL, R. M., AND CLEARY, J. G. Arithmetic coding for data compression. *Commun. ACM 30*, 6 (June, 1987), 520–540.

[ZL78] ZIV, J. AND LEMPEL, A. Compression of individual sequences via variable- rate coding. *IEEE Trans. Inf. Theory 24*, 5 (Sept., 1978), 530–536.